Fundamentos de
lógica digital
con
diseño VHDL

Segunda edición

Stephen Brown
Zvonko Vranesic

Mc Graw Hill

INCLUYE CD-ROM

En esta obra se enseñan las técnicas básicas de diseño de circuitos lógicos, con especial atención en la síntesis de los circuitos y la implementación de los circuitos en chips reales. Los conceptos fundamentales se ilustran mediante ejemplos sencillos, fáciles de entender. Además, se aplica un enfoque modular para mostrar cómo se diseñan los circuitos más grandes. VHDL se usa para demostrar cómo se definen en un lenguaje de descripción de hardware los bloques constructores básicos y los sistemas más grandes, con lo que se producen diseños que pueden implementarse con modernas herramientas de software de diseño asistido por computadora (CAD).

La exposición basada en VHDL es de suma utilidad, ya que permite que el lector se involucre rápidamente con los diseños reales. Por añadidura, el libro puede usarse con cualquier sistema CAD para diseño e implementación de circuitos lógicos. Asimismo, para que el usuario obtenga modernas herramientas CAD, se incluye un disco compacto que contiene Quartus II, de Altera, un programa CAD que permite que los diseños escritos en VHDL se traduzcan de manera automática en arreglos de compuertas de campos programables (FPGA) y en dispositivos lógicos programables complejos (CPLD). El usuario será capaz de:

- Ingresar un diseño en el sistema CAD
- Compilar el diseño en un dispositivo específico
- Simular la funcionalidad y la sincronización del circuito resultante
- Implementar el diseño en dispositivos reales (en las instalaciones del laboratorio escolar)

VHDL es un lenguaje complejo, de modo que su introducción en el texto es gradual. Cada característica se presenta conforme se vuelve pertinente para el análisis de los circuitos. A fin de enseñar a los estudiantes el uso del programa Quartus II, en el libro se incluyen tres tutoriales, además de una exposición de todo el flujo CAD. Esta herramienta se presenta en el CD en el idioma inglés.

FUNDAMENTOS
DE
LÓGICA DIGITAL CON DISEÑO VHDL

FUNDAMENTOS DE LÓGICA DIGITAL CON DISEÑO VHDL

SEGUNDA EDICIÓN

Stephen Brown y Zvonko Vranesic
Departamento de Ingeniería Eléctrica y Computación
University of Toronto

Traducción
Lorena Peralta Rosales
Víctor Campos Olguín
Traductores profesionales

Revisión técnica
Jorge Valeriano Assem
Coordinador de la carrera de Ingeniería en Computación
Universidad Nacional Autónoma de México

Felipe Antonio Trujillo Fernández
Profesor del Departamento de Ingenierías
Universidad Iberoamericana, Campus Santa Fe

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA
MADRID • NUEVA YORK • SAN JUAN • SANTIAGO
AUCKLAND • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Director editorial: Ricardo A. del Bosque Alayón
Editor sponsor: Pablo Eduardo Roig Vázquez
Editora de desarrollo: Paula Montaño González
Supervisor de producción: Zeferino García García

Fundamentos de lógica digital con diseño VHDL
Segunda edición

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin la autorización escrita del editor.

McGraw-Hill Interamericana

DERECHOS RESERVADOS © 2006 respecto a la primera edición en español por
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.
*A Subsidiary of The **McGraw-Hill** Companies, Inc.*
Edificio Punta Santa Fe
Prolongación Paseo de la Reforma 1015, Torre A
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN 970-10-5609-4

Traducido de la segunda edición de: FUNDAMENTALS OF DIGITAL LOGIC WITH VHDL DESIGN
Copyright © MMV by The McGraw-Hill Companies, Inc. All rights reserved.
Previous edition © 2000.
ISBN 0-07-246085-7

1234567890 09875432106

Impreso en México Printed in México

Para Susan y Anne

ACERCA DE LOS AUTORES

Stephen Brown obtuvo la licenciatura en ingeniería eléctrica por la Universidad de New Brunswick, Canadá, y la maestría y el doctorado en ingeniería eléctrica por la Universidad de Toronto. En 1992 se unió al cuerpo académico de esta última institución, donde ahora es profesor asociado del Departamento de Ingeniería Eléctrica y de Cómputo. También es director de desarrollo de software en el Altera Toronto Technology Center.

Sus intereses de investigación incluyen la tecnología VLSI de campos programables y la arquitectura de computadoras. En 1992 obtuvo el premio doctoral del Canadian Natural Sciences and Engineering Research Council por la mejor tesis de doctorado en Canadá.

Ganó cuatro premios por excelencia en la enseñanza de cursos de ingeniería eléctrica, ingeniería de cómputo y ciencias computacionales. Es coautor de otros dos libros: *Fundamentals of Digital Logic with Verilog Design* y *Field-Programmable Gate Arrays*.

Zvonko Vranesic obtuvo la licenciatura, la maestría y el doctorado, todos en ingeniería eléctrica, por la Universidad de Toronto. De 1963 a 1965 trabajó como ingeniero de diseño en la Northern Electric Co. Ltd., en Bramalea, Ontario. En 1968 se unió al cuerpo académico de la Universidad de Toronto, donde ahora es profesor de los departamentos de Ingeniería Eléctrica y Cómputo, y de Ciencias Computacionales. Durante el año académico 1978-1979 fue profesor huésped en la Universidad de Cambridge, Inglaterra, y durante 1984-1985 estuvo en la Universidad de París, 6. De 1995 a 2000 fungió como jefe de la División de Ciencias de Ingeniería en la Universidad de Toronto. También participó en investigación y el desarrollo en la Altera Toronto Technology Center.

Sus actuales intereses de investigación abarcan arquitectura de computadoras, tecnología VLSI de campos programables y sistemas de lógica multivaluada.

Es coautor de otros cuatro libros: *Computed Organization,* 5a. ed.; *Fundamentals of Digital Logic with Verilog Design*; *Microcomputer Structures* y *Field-Programmable Gate Arrays*. En 1990 recibió la Wighton Fellowship por sus "innovadores e inconfundibles aportes en la enseñanza de laboratorio de estudiantes de licenciatura". En 2004 recibió el Faculty Teaching Award de la Facultad de Ciencias Aplicadas e Ingeniería por la Universidad de Toronto.

Ha representado a Canadá en varias competencias de ajedrez. Posee el título de maestro internacional.

PREFACIO

Este libro está diseñado para un curso de introducción al diseño de lógica digital, que es un curso básico en la mayor parte de los programas de ingeniería eléctrica y cómputo. Un buen diseñador de circuitos lógicos digitales debe entender a cabalidad los conceptos básicos y manejar con destreza las herramientas de diseño asistido por computadora (CAD). Por tanto, el propósito de esta obra es ofrecer el equilibrio deseable entre la enseñanza de los conceptos básicos y la aplicación práctica con herramientas CAD. Para facilitar el proceso de aprendizaje, el software CAD necesario se incluye como parte integral del libro.

Son dos las metas principales de este texto: (1) enseñar a los estudiantes los conceptos fundamentales del diseño digital manual clásico y (2) ilustrar claramente cómo se diseñan hoy en día los circuitos digitales mediante herramientas CAD. Aun cuando los diseñadores modernos ya no siguen las técnicas manuales, salvo en raras circunstancias, nuestra motivación para enseñarlas estriba en que los estudiantes pueden obtener nociones más bien intuitivas de cómo funcionan los circuitos digitales. Además, las técnicas manuales ilustran bien los tipos de manipulaciones que las herramientas CAD realizan, con lo que se brinda a los estudiantes los conocimientos suficientes para que aprecien los beneficios ofrecidos por la automatización del diseño. A lo largo del libro se exponen conceptos básicos por medio de técnicas manuales y métodos modernos basados en herramientas CAD. Tras establecer los conceptos fundamentales, se brindan ejemplos más complejos, apoyados en las herramientas CAD. Por ende, el énfasis se centra en los métodos modernos de diseño a fin de ilustrar cómo se realiza actualmente el diseño digital.

TECNOLOGÍA Y APOYO CAD

En el libro analizamos las modernas tecnologías de implementación de circuitos digitales. Ponemos el acento en los dispositivos lógicos programables (PLD), que son, por dos razones, la tecnología más apropiada para usar en un libro de texto. La primera es que los PLD se usan mucho en la práctica y están disponibles para casi todos los tipos de diseños de circuitos digitales. De hecho, en algún punto de su carrera los estudiantes tienen más probabilidad de involucrarse en diseños basados en PLD que en cualquier otra tecnología. La segunda razón: los circuitos se implementan en PLD mediante programación del usuario final. Por tanto, es posible ofrecer a los estudiantes la oportunidad de implementar en el laboratorio y en chips reales los ejemplos de diseño expuestos en el libro. También pueden simular el comportamiento de sus circuitos diseñados en sus propias computadoras. Como dispositivos destino de los diseños hemos usado los dos tipos más populares de PLD: dispositivos lógicos programables complejos (CPLD) y arreglos de compuertas de campos programables (FPGA).

Nuestro soporte en CAD se basa en el programa Quartus II, de Altera, que ofrece la traducción automática de un diseño en CPLD y FPGA de Altera, que se encuentran entre los PLD más ampliamente usados en la industria. Las características de Quartus II particularmente atractivas para los propósitos de este texto son:

- Se trata de un producto comercial. La versión que se incluye en el libro soporta todas las características principales del producto. Los estudiantes serán capaces de ingresar fácilmente

un diseño en el sistema, compilarlo en un dispositivo elegido (la elección del dispositivo puede cambiarse en cualquier momento), simular la funcionalidad y la sincronización detallada del circuito resultante y, si la escuela posee instalaciones de laboratorio, implementar los diseños en dispositivos reales.

- Ofrece el ingreso del diseño usando lenguajes de descripción de hardware (HDL) o captura esquemática. En el libro se hace énfasis en el diseño basado en HDL, pues es el método de diseño más eficiente en la práctica. En los ejemplos se describe con detalle el lenguaje de VHDL según el estándar del IEEE y se le usa de manera extensa. El sistema CAD incluido con el libro tiene un compilador de VHDL que permite al estudiante crear automáticamente circuitos a partir de código de VHDL e implementarlos en chips reales.
- Un diseño puede dirigirse de manera automática a varios tipos de dispositivos, lo que permite ilustrar las formas en las que la arquitectura del dispositivo blanco afectan el circuito de un diseñador.
- Puede usarse en la mayor parte de las computadoras populares. La versión de Quartus II ofrecida con el libro se ejecuta en computadoras que usan Microsoft Windows NT, 2000 o XP. Sin embargo, mediante el programa universitario de Altera, el software también está disponible para otras máquinas, como las estaciones de trabajo SUN y HP.

Con cada copia de esta obra se incluye un disco compacto de Quartus II. El uso del software está totalmente integrado en el libro, de modo que los estudiantes pueden probar, de primera mano, todos los ejemplos de diseño. Para enseñarles a utilizar este software, en el libro se incluyen tres tutoriales que avanzan progresivamente.

ÁMBITO DEL LIBRO

En el capítulo 1 se presenta una introducción general al proceso de diseño de sistemas digitales. Se abordan los pasos principales de él y se explica cómo usar las herramientas CAD para automatizar una buena parte de las tareas requeridas.

En el capítulo 2 se introducen los aspectos básicos de los circuitos lógicos y se muestra cómo usar el álgebra booleana para representarlos. También se ofrece al lector un primer vistazo de VHDL como ejemplo de un lenguaje de descripción de hardware que sirve para especificar los circuitos lógicos.

Los aspectos electrónicos de los circuitos digitales se presentan en el capítulo 3, donde se muestra cómo se construyen las compuertas básicas mediante transistores. Asimismo se presentan varios factores que influyen en el desempeño del circuito. El énfasis se centra en las tecnologías más recientes, con acento especial en la tecnología CMOS y los dispositivos lógicos programables.

En el capítulo 4 se aborda la síntesis de los circuitos combinacionales. Se abarcan todos los aspectos del proceso de síntesis, que comienza con un diseño inicial y desarrolla los pasos de optimación necesarios para generar el circuito final buscado. Se muestra cómo utilizar las herramientas CAD para este propósito.

El capítulo 5 se centra en los circuitos que realizan operaciones aritméticas. Se parte de la explicación de cómo se representan los números en los sistemas digitales y luego se muestra la manera de manipularlos por medio de circuitos lógicos. En este capítulo se ilustra cómo usar VHDL para especificar la funcionalidad deseada y el modo en que las herramientas CAD proporcionan un mecanismo para desarrollar los circuitos buscados. Elegimos introducir las representaciones de los números en este punto, en vez de al comienzo del libro, para hacer la

explicación más significativa e interesante, pues inmediatamente pueden ofrecerse ejemplos de cómo procesar la información numérica con circuitos reales.

En el capítulo 6 se presentan los circuitos combinacionales que sirven como bloques constructores. En él se incluyen circuitos codificadores, decodificadores y multiplexores, circuitos muy prácticos para ilustrar la aplicación de muchos constructores de VHDL, al tiempo que brindan al lector la oportunidad de descubrir características más avanzadas de ese lenguaje.

Los elementos de almacenamiento se presentan en el capítulo 7. Se explica el uso de los flip-flops para realizar estructuras regulares, como registros de corrimiento y contadores. Se incluyen diseños especificados con VHDL de tales estructuras. En el capítulo también se muestra cómo diseñar circuitos más grandes, entre ellos un procesador simple.

En el capítulo 8 se tratan con detalles los circuitos secuenciales síncronos (máquinas de estado finito); se explica su comportamiento y se desarrollan técnicas de diseño práctico tanto manual como automatizado.

Los circuitos secuenciales asíncronos se abordan en el capítulo 9. Si bien la exposición no es exhaustiva, ofrece una buena indicación de las características principales de esos circuitos. Aun cuando los circuitos asíncronos no se usan mucho en la práctica, es preciso estudiarlos, pues son un excelente vehículo para conocer mejor el funcionamiento de los circuitos digitales en general: ilustran las consecuencias de los retrasos de propagación y las condiciones de carrera que pueden ser inherentes en la estructura de un circuito.

En el capítulo 10 se presenta una explicación de varios conflictos prácticos que surgen durante el diseño de los sistemas reales. Se destacan los problemas que se encuentran con más frecuencia en la práctica, al tiempo que se indica cómo superarlos. Los ejemplos de circuitos más grandes ilustran un enfoque jerárquico en el diseño de los sistemas digitales. Se presenta un código de VHDL completo de tales circuitos.

En el capítulo 11 se introduce el tema de las pruebas. Un diseñador de circuitos lógicos ha de estar consciente de la necesidad de probar los circuitos y debe conocer al menos los aspectos más básicos de ello.

En el capítulo 12 se presenta el flujo completo del diseño asistido por computadora que el diseñador experimenta cuando diseña, implementa y pone a prueba un circuito digital.

En el apéndice A se ofrece un resumen completo de las características de VHDL. Aunque el uso de este lenguaje se integra a lo largo de la obra, en este apéndice se proporciona una referencia práctica que el lector puede consultar de vez en vez al escribir un código de VHDL.

Los apéndices B, C y D contienen una secuencia de tutoriales acerca de las herramientas CAD de Quartus II. Este material es adecuado para el autoestudio; muestra al estudiante, paso a paso, cómo usar el software CAD incluido con el libro.

En el apéndice E se brinda información detallada de los dispositivos empleados en los ejemplos del texto.

QUÉ SE PUEDE CUBRIR EN UN CURSO

Todo el material del libro puede cubrirse en dos cursos de un trimestre cada uno. Una buena cobertura del material más importante puede lograrse en un curso de un solo semestre, o incluso en uno de un trimestre. Esto sólo es posible si el instructor no emplea demasiado tiempo en la enseñanza de las complejidades de VHDL y las herramientas CAD. Para poner en práctica este enfoque, el material de VHDL se organizó en un estilo modular que conduce al autoestudio. La experiencia de los autores en la enseñanza a diferentes tipos de estudiantes en la Universidad de

Toronto demuestra que el instructor puede emplear de dos a tres horas de clase de VHDL, y centrarse principalmente en la especificación de los circuitos secuenciales. Los ejemplos de VHDL dados en el libro son autoexplicativos, y los estudiantes pueden comprenderlos sin problemas. Más aún, no es necesario que el instructor enseñe cómo usar las herramientas CAD, pues los tutoriales de Quartus II presentados en los apéndices B, C y D son adecuados para ello.

El libro también es adecuado para un curso de diseño lógico que no incluya VHDL. Sin embargo, cierto conocimiento de ese lenguaje, incluso en un nivel rudimentario, será provechoso para los estudiantes, y es una gran preparación para un empleo como ingeniero de diseño.

Curso de un semestre

Un punto de partida natural para las clases es el capítulo 2. El material del capítulo 1 es una introducción general que sirve como motivación de por qué los circuitos lógicos son importantes e interesantes; los estudiantes pueden leer y comprender este material con facilidad.

En las clases debe cubrirse el material siguiente:

- Capítulo 2: todas las secciones.
- Capítulo 3: secciones 3.1 a 3.7. Además, es útil cubrir las secciones 3.8 y 3.9 si los estudiantes tienen algún conocimiento básico de circuitos eléctricos.
- Capítulo 4: secciones 4.1 a 4.7 y sección 4.12.
- Capítulo 5: secciones 5.1 a 5.5.
- Capítulo 6: todas las secciones.
- Capítulo 7: todas las secciones.
- Capítulo 8: secciones 8.1 a 8.9.

Si el tiempo lo permite, también sería muy útil cubrir las secciones 9.1 a 9.3 y la sección 9.6 del capítulo 9, así como uno o dos ejemplos del capítulo 10.

Curso de un trimestre

En un curso de un trimestre puede cubrirse el material siguiente:

- Capítulo 2: todas las secciones.
- Capítulo 3: secciones 3.1 a la 3.3.
- Capítulo 4: secciones 4.1 a la 4.5 y la sección 4.12.
- Capítulo 5: secciones 5.1 a la 5.3 y la sección 5.5.
- Capítulo 6: todas las secciones.
- Capítulo 7: secciones 7.1 a la 7.10 y la sección 7.13.
- Capítulo 8: secciones 8.1 a la 8.5.

UN ENFOQUE MÁS TRADICIONAL

En el material de los capítulos 2 y 4 se expone el álgebra booleana, los circuitos lógicos combinacionales y las técnicas básicas de minimización. En el capítulo 2 se ofrece la explicación ini-

cial de estos temas empleando únicamente compuertas AND, OR, NOT, NAND y NOR. Luego, en el capítulo 3 se abordan los detalles de la tecnología de implementación, antes de proceder con las técnicas de síntesis y otros tipos de compuertas en el capítulo 4. El material de este capítulo se aprecia mejor si los estudiantes entienden las razones tecnológicas para la existencia de las compuertas NAND, NOR y XOR, y los diversos dispositivos lógicos programables.

Un instructor que favorezca un enfoque más tradicional puede cubrir los capítulos 2 y 4 en sucesión. Para comprender el uso de las compuertas NAND, NOR y XOR sólo se precisa que se las defina funcionalmente.

VHDL

VHDL es un lenguaje complejo, y algunos instructores lo consideran demasiado difícil de comprender para los estudiantes de primeros ciclos. Hemos tenido en cuenta este conflicto y hemos procurado resolverlo. No es necesario explicar todo el VHDL. En el libro se presentan sus constructores importantes, útiles para el diseño y la síntesis de circuitos lógicos. Se omiten muchos otros constructores, como los que sólo tienen significado cuando el lenguaje se usa con propósitos de simulación. El material de VHDL se introduce gradualmente, y las características más avanzadas sólo se presentan donde su empleo puede demostrarse en el diseño de circuitos relevantes.

En el libro se incluyen más de 150 ejemplos de código de VHDL, con los que se ilustra cómo usar ese lenguaje para describir un amplio repertorio de circuitos lógicos, desde los que sólo contienen unas cuantas compuertas hasta los que representan sistemas digitales como un procesador simple.

PROBLEMAS RESUELTOS

En cada capítulo se incluyen ejemplos de problemas resueltos, los cuales muestran cómo resolver los típicos problemas de tarea en casa.

PROBLEMAS DE TAREA

El libro proporciona más de 400 problemas de tarea; en la parte final se ofrecen las respuestas a algunos de ellos. Las soluciones de todos los problemas están a disposición de los instructores en el *Manual de soluciones* que acompaña a la obra.

LABORATORIO

El libro puede usarse para un curso que no incluya ejercicios de laboratorio, en cuyo caso los estudiantes pueden adquirir útil experiencia práctica al simular la operación de sus circuitos diseñados empleando las herramientas CAD incluidas en el libro. Si hay un laboratorio disponible, entonces varios de los ejemplos de diseño presentados se adecuan bien para realizar experimentos. En la página en Internet de los autores están disponibles experimentos adicionales.

AGRADECIMIENTOS

Queremos expresar nuestro agradecimiento a las personas que nos ayudaron durante la preparación del libro. Kelly Chan ayudó con la preparación técnica del manuscrito. Dan Vranesic produjo una cantidad sustancial de gráficos e imágenes. Él y Deshanand Singh también ayudaron con la preparación del manual de soluciones. Tom Czajkowski auxilió en la comprobación de las respuestas a algunos problemas. Los siguientes revisores hicieron críticas constructivas y realizaron numerosas sugerencias para mejorar el texto: William Barnes, New Jersey Institute of Technology; Thomas Bradicich, North Carolina State University; James Clark, McGill University; Stephen DeWeerth, Georgia Institute of Technology; Clay Gloster, Jr., North Carolina State University (Raleigh); Carl Hamacher, Queen's University; Vincent Heuring, University of Colorado; Yu Hen Hu, University of Wisconsin; Wei-Ming Lin, University of Texas (Austin); Wayne Loucks, University of Waterloo; Chris Myers, University of Utah; Vojin Oklobdzija, University of California (Davis); James Palmer, Rochester Institute of Technology; Gandhi Puvvada, University of Southern California; Teodoro Robles, Milwaukee School of Engineering; Tatyana Roziner, Boston University; Rob Rutenbar, Carnegie Mellon University; Eric Schwartz, University of Florida; Wen-Tsong Shiue, Oregon State University; Charles Silio, Jr., University of Maryland; Scott Smith, University of Missouri (Rolla); Arun Somani, Iowa State University; Bernard Svihel, University of Texas (Arlington); y Zeljko Zilic, McGill University.

Agradecemos a Altera Corporation el proporcionar el sistema Quartus II, en especial a Chris Balough, Misha Burich, Joe Hanson, Mike Phipps y Tim Southgate. El apoyo del personal de McGraw-Hill fue ejemplar. Verdaderamente apreciamos la ayuda de Dawn Bercier, Melinda Bilecki, Kay Brimeyer, Michaela Graham, Betsy Jones, Kara Kudronowicz, Carlise Paulson, Eric Weber y Michelle Whitaker.

Stephen Brown y Zvonko Vranesic

CONTENIDO

Capítulo 1
CONCEPTOS DE DISEÑO 1

1.1 Hardware digital 2
1.1.1 Chips estándar 4
1.1.2 Dispositivos lógicos programables 4
1.1.3 Chips diseñados a la medida 5
1.2 El proceso de diseño 6
1.3 Diseño de hardware digital 8
1.3.1 Ciclo de diseño básico 8
1.3.2 Estructura de una computadora 9
1.3.3 Diseño de una unidad de hardware digital 12
1.4 Diseño de circuitos lógicos en este libro 16
1.5 Teoría y práctica 16
Bibliografía 17

Capítulo 2
INTRODUCCIÓN A LOS CIRCUITOS LÓGICOS 19

2.1 Variables y funciones 20
2.2 Inversión 23
2.3 Tablas de verdad 24
2.4 Compuertas lógicas y circuitos 25
2.4.1 Análisis de una red lógica 27
2.5 Álgebra booleana 29
2.5.1 Los diagramas de Venn 33
2.5.2 Notación y terminología 35
2.5.3 Precedencia de las operaciones 37
2.6 La síntesis con compuertas AND, OR y NOT 37
2.6.1 Formas de productos de sumas y sumas de productos 39
2.7 Circuitos lógicos NAND y NOR 45
2.8 Ejemplos de diseño 50
2.8.1 Control de luz de tres vías 50
2.8.2 Circuito multiplexor 51
2.9 Introducción a las herramientas CAD 54
2.9.1 Ingreso del diseño 54
2.9.2 Síntesis 56
2.9.3 Simulación funcional 57
2.9.4 Diseño físico 57
2.9.5 Simulación de tiempo 57
2.9.6 Configuración de chip 58
2.10 Introducción a VHDL 58
2.10.1 Representación de señales digitales en VHDL 60
2.10.2 Cómo escribir código sencillo en VHDL 60
2.10.3 Cómo *no* escribir código de VHDL 62
2.11 Comentarios finales 63
2.12 Ejemplos de problemas resueltos 64
Problemas 67
Bibliografía 72

Capítulo 3
TECNOLOGÍA DE IMPLEMENTACIÓN 73

3.1 Interruptores de transición 75
3.2 Compuertas lógicas NMOS 78
3.3 Compuertas lógicas CMOS 81
3.3.1 Velocidad de los circuitos de compuerta lógica 87
3.4 Sistema de lógica negativa 87
3.5 Chips estándar 91
3.5.1 Chips estándar de la serie 7400 91
3.6 Dispositivos lógicos programables 94
3.6.1 Arreglo lógico programable (PLA) 94
3.6.2 Lógica de arreglo programable 97
3.6.3 Programación de PLA y PAL 99
3.6.4 Dispositivos lógicos programables complejos (CPLD) 101
3.6.5 Arreglos de compuertas de campos programables 105
3.6.6 Uso de herramientas CAD para implementar circuitos en CPLD y FPGA 110
3.6.7 Aplicaciones de los CPLD y FPGA 110
3.7 Chips diseñados a la medida, celdas estándar y arreglos de compuertas 110
3.8 Aspectos prácticos 114
3.8.1 Fabricación y comportamiento de los MOSFET 114
3.8.2 MOSFET con resistencia de encendido (on-resistance) 117
3.8.3 Niveles de voltaje en compuertas lógicas 118
3.8.4 Margen de ruido 119
3.8.5 Operación dinámica de las compuertas lógicas 121

3.8.6 Disipación de potencia en las compuertas lógicas 124
3.8.7 Paso de 1 y 0 mediante interruptores de transistor 126
3.8.8 Factores de carga de entrada y de salida en las compuertas lógicas 128
3.9 Compuertas de transmisión 134
3.9.1 Compuertas OR exclusiva 135
3.9.2 Circuito multiplexor 136
3.10 Detalles de implementación para SPLD, CPLD y FPGA 136
3.10.1 Implementación en FPGA 142
3.11 Comentarios finales 145
3.12 Ejemplos de problemas resueltos 145
Problemas 153
Bibliografía 162

Capítulo 4
IMPLEMENTACIÓN OPTIMIZADA DE FUNCIONES LÓGICAS 163

4.1 Mapa de Karnaugh 164
4.2 Estrategia de minimización 172
4.2.1 Terminología 173
4.2.2 Procedimiento de minimización 175
4.3 Minimización de formas de producto de sumas 178
4.4 Funciones especificadas de manera incompleta 180
4.5 Circuitos de salida múltiple 182
4.6 Síntesis multinivel 185
4.6.1 Factorización 186
4.6.2 Descomposición funcional 190
4.6.3 Circuitos NAND y NOR multinivel 195
4.7 Análisis de circuitos multinivel 196
4.8 Representación cúbica 203
4.8.1 Cubos e hipercubos 203
4.9 Un método tabular para minimización 207
4.9.1 Generación de implicantes primos 208
4.9.2 Determinación de una cobertura mínima 209
4.9.3 Resumen del método tabular 215
4.10 Una técnica cúbica de minimización 216
4.10.1 Determinación de los implicantes primos esenciales 218
4.10.2 Procedimiento completo para hallar una cobertura mínima 220
4.11 Consideraciones prácticas 223
4.12 Ejemplos de circuitos sintetizados a partir de código de VHDL 224
4.13 Comentarios finales 228
4.14 Ejemplos de problemas resueltos 229
Problemas 237
Bibliografía 242

Capítulo 5
REPRESENTACIÓN DE NÚMEROS Y CIRCUITOS ARITMÉTICOS 245

5.1 Representación numérica posicional 246
5.1.1 Enteros sin signo 246
5.1.2 Conversión entre sistemas decimal y binario 247
5.1.3 Representaciones octal y hexadecimal 248
5.2 Suma de números sin signo 250
5.2.1 Sumador completo descompuesto 254
5.2.2 Sumador con acarreo en cascada 254
5.2.3 Ejemplo de diseño 255
5.3 Números con signo 256
5.3.1 Números negativos 256
5.3.2 Suma y resta 260
5.3.3 Unidad sumadora y restadora 264
5.3.4 Esquema de complemento a la base (raíz) 265
5.3.5 Desbordamiento aritmético 269
5.3.6 Problemas de rendimiento 270
5.4 Sumadores veloces 271
5.4.1 Sumador con acarreo de adelanto 271
5.5 Diseño de circuitos aritméticos con el uso de herramientas CAD 278
5.5.1 Diseño de circuitos aritméticos con el uso de captura esquemática 278
5.5.2 Diseño de circuitos aritméticos con VHDL 281
5.5.3 Representación de números en código de VHDL 284
5.5.4 Instrucciones de asignación aritmética 285
5.6 Multiplicación 289
5.6.1 Arreglo multiplicador para números sin signo 291
5.6.2 Multiplicación de números con signo 292
5.7 Otras representaciones numéricas 293
5.7.1 Números con punto fijo 293
5.7.2 Números con punto flotante 295
5.7.3 Representación decimal codificado en binario 297

5.8 Código de caracteres ASCII 301
5.9 Ejemplos de problemas resueltos 304
Problemas 310
Bibliografía 314

Capítulo 6
BLOQUES CONSTRUCTORES DE CIRCUITOS COMBINACIONALES 315

6.1 Multiplexores 316
6.1.1 Síntesis de funciones lógicas mediante multiplexores 321
6.1.2 Síntesis de multiplexores mediante la expansión de Shannon 324
6.2 Decodificadores 329
6.2.1 Demultiplexores 333
6.3 Codificadores 335
6.3.1 Codificadores binarios 335
6.3.2 Codificadores de prioridad 336
6.4 Convertidores de código 337
6.5 Circuitos de comparación aritmética 338
6.6 VHDL para circuitos combinacionales 339
6.6.1 Instrucciones de asignación 339
6.6.2 Asignación de señal seleccionada 340
6.6.3 Asignación de señal condicional 340
6.6.4 Instrucciones de generación 348
6.6.5 Instrucciones de asignación concurrente y secuencial 350
6.6.6 Instrucción PROCESS 350
6.6.7 Instrucción CASE 356
6.6.8 Operadores de VHDL 359
6.7 Comentarios finales 363
6.8 Ejemplos de problemas resueltos 364
Problemas 372
Bibliografía 377

Capítulo 7
FLIP-FLOPS, REGISTROS, CONTADORES Y UN PROCESADOR SIMPLE 379

7.1 El latch básico 381
7.2 Latch SR asíncrono 383
7.2.1 Latch SR asíncrono con compuertas NAND 385
7.3 Latch D asíncrono 386
7.3.1 Efectos de los retrasos de propagación 388
7.4 Flip-flops D maestro-esclavo y disparado por flanco 389
7.4.1 Flip-flop D maestro-esclavo 389
7.4.2 Flip-flop D disparado por flanco 389
7.4.3 Flip-flops D con Clear y Preset 393
7.5 Flip-flop T 394
7.5.1 Flip-flops configurables 397
7.6 Flip-flop JK 397
7.7 Resumen de terminología 398
7.8 Registros 399
7.8.1 Registro de corrimiento 399
7.8.2 Registro de corrimiento con acceso en paralelo 400
7.9 Contadores 400
7.9.1 Contadores asíncronos 401
7.9.2 Contadores síncronos 404
7.9.3 Contadores con carga en paralelo 408
7.10 Reset síncrono 408
7.11 Otros tipos de contadores 412
7.11.1 Contador BCD 412
7.11.2 Contador en anillo 413
7.11.3 Contador Johnson 415
7.11.4 Comentarios sobre el diseño del contador 415
7.12 Uso de elementos de almacenamiento con herramientas CAD 416
7.12.1 Inclusión de elementos de almacenamiento en esquemas 416
7.12.2 Uso de constructores de VHDL para elementos de almacenamiento 417
7.13 Uso de registros y contadores con herramientas CAD 423
7.13.1 Inclusión de registros y contadores en esquemas 423
7.13.2 Registros y contadores en código de VHDL 426
7.13.3 Uso de instrucciones secuenciales de VHDL para registros y contadores 427
7.14 Ejemplos de diseño 435
7.14.1 Estructura de bus 435
7.14.2 Procesador simple 449
7.14.3 Contador de tiempo de reacción 460
7.14.4 Código de nivel de transferencia de registros (RTL) 466
7.15 Comentarios finales 466
7.16 Ejemplos de problemas resueltos 467
Problemas 471
Bibliografía 477

Capítulo 8
CIRCUITOS SÍNCRONOS SECUENCIALES 479

8.1 Pasos básicos de diseño 481
8.1.1 Diagrama de estado 481
8.1.2 Tabla de estado 483
8.1.3 Asignación de estados 483
8.1.4 Elección de flip-flops y derivación de las expresiones de estado siguiente y de salida 485
8.1.5 Diagrama de tiempo 486
8.1.6 Resumen de los pasos de diseño 488
8.2 El problema de la asignación de estados 491
8.2.1 Codificación de 1 activo 494
8.3 Modelo de estado tipo Mealy 496
8.4 Diseño de máquinas de estado finito con herramientas CAD 501
8.4.1 Código de VHDL para FSM tipo Moore 502
8.4.2 Síntesis del código de VHDL 504
8.4.3 Simulación y prueba del circuito 506
8.4.4 Un estilo alterno de código de VHDL 507
8.4.5 Resumen de los pasos de diseño cuando se usan herramientas CAD 507
8.4.6 Especificación de la asignación de estados en el código de VHDL 509
8.4.7 Especificación de FSM tipo Mealy con VHDL 511
8.5 Ejemplo de sumador serial 513
8.5.1 FSM tipo Mealy para sumador serial 514
8.5.2 FSM tipo Moore para el sumador serial 516
8.5.3 Código de VHDL para el sumador serial 518
8.6 Minimización de estados 522
8.6.1 Procedimiento de minimización por partición 524
8.6.2 FSM especificadas de manera incompleta 531
8.7 Diseño de un contador utilizando el enfoque del circuito secuencial 533
8.7.1 Diagrama de estado y tabla de estado para un contador módulo 8 533
8.7.2 Asignación de estados 534
8.7.3 Implementación utilizando flip-flops D 535
8.7.4 Implementación utilizando flip-flops JK 536
8.7.5 Ejemplo. Un contador diferente 541
8.8 FSM como un circuito árbitro 543
8.8.1 Implementación del circuito árbitro 547
8.8.2 Minimización de los retrasos de salida para una FSM 550
8.8.3 Resumen 551
8.9 Análisis de los circuitos secuenciales síncronos 551
8.10 Cartas de la máquina algorítmica de estados (cartas ASM) 555
8.11 Modelo formal para circuitos secuenciales 559
8.12 Comentarios finales 560
8.13 Ejemplos de problemas resueltos 561
Problemas 570
Bibliografía 574

Capítulo 9
CIRCUITOS SECUENCIALES ASÍNCRONOS 577

9.1 Comportamiento asíncrono 578
9.2 Análisis de los circuitos asíncronos 582
9.3 Síntesis de los circuitos asíncronos 590
9.4 Reducción de estados 603
9.5 Asignación de estados 618
9.5.1 Diagrama de transición 621
9.5.2 Cómo aprovechar las combinaciones de estado siguiente sin especificar 624
9.5.3 Asignación de estados usando variables de estado adicionales 628
9.5.4 Asignación de estados con codificación de 1 activo 633
9.6 Riesgos 634
9.6.1 Riesgos estáticos 635
9.6.2 Riesgos dinámicos 639
9.6.3 Relevancia de los riesgos 640
9.7 Un ejemplo de diseño completo 642
9.7.1 El controlador de la máquina expendedora 642
9.8 Comentarios finales 647
9.9 Ejemplos de problemas resueltos 649
Problemas 657
Bibliografía 661

Capítulo 10
DISEÑO DE SISTEMAS DIGITALES 663

10.1 Circuitos de bloque de construcción 664
10.1.1 Flip-flops y registros con entradas enable 664

10.1.2 Registros de corrimiento con entradas enable 666
10.1.3 Memoria estática de acceso aleatorio (SRAM) 668
10.1.4 Bloques SRAM en PLD 673
10.2 Ejemplos de diseño 673
10.2.1 Un circuito de conteo de bits 673
10.2.2 Información de sincronización esbozada en la carta ASM 675
10.2.3 Multiplicador de corrimiento y suma 677
10.2.4 Divisor 686
10.2.5 Media aritmética 696
10.2.6 Operación de ordenación 702
10.3 Sincronización del reloj 713
10.3.1 Desviación del reloj 713
10.3.2 Parámetros de sincronización de los flip-flops 714
10.3.3 Entradas asíncronas a los flip-flops 717
10.3.4 Eliminación de rebotes en interruptores 718
10.4 Comentarios finales 718
Problemas 720
Bibliografía 724

Capítulo 11
PRUEBA DE LOS CIRCUITOS LÓGICOS 725

11.1 Modelo de fallas 726
11.1.1 Modelo de atascamiento *(stuck-at)* 726
11.1.2 Fallas individuales y múltiples 727
11.1.3 Circuitos CMOS 727
11.2 Complejidad de un conjunto de pruebas 727
11.3 Sensibilización de trayectorias 729
11.3.1 Detección de una falla específica 731
11.4 Circuitos con la estructura de árbol 733
11.5 Pruebas aleatorias 734
11.6 Pruebas de circuitos secuenciales 737
11.6.1 Diseño para la aplicación de pruebas 737
11.7 Prueba automatizada integrada 741
11.7.1 Observador de bloques lógicos integrado 745
11.7.2 Análisis de firmas 747
11.7.3 Boundary Scan 748
11.8 Tarjetas de circuitos impresos 748
11.8.1 Pruebas de las PCB 750
11.8.2 Instrumentación 751
11.9 Comentarios finales 752
Problemas 752
Bibliografía 755

Capítulo 12
HERRAMIENTAS DE DISEÑO ASISTIDO POR COMPUTADORA 757

12.1 Síntesis 758
12.1.1 Generación de la lista de redes 758
12.1.2 Optimación de compuertas 758
12.1.3 Mapeo de tecnología 760
12.2 Diseño físico 764
12.2.1 Colocación 767
12.2.2 Enrutamiento 768
12.2.3 Análisis de tiempo estático 769
12.3 Comentarios finales 771
Bibliografía 771

Apéndice A
REFERENCIA DE VHDL 773

A.1 Documentación en el código de VHDL 774
A.2 Objetos de datos 774
A.2.1 Nombres de objetos de datos 774
A.2.2 Valores y números del objeto de datos 774
A.2.3 Objetos de datos SIGNAL 775
A.2.4 Tipos BIT y BIT_VECTOR 775
A.2.5 Tipos STD_LOGIC y STD_LOGIC_VECTOR 776
A.2.6 Tipos STD_ULOGIC 776
A.2.7 Tipos SIGNED y UNSIGNED 777
A.2.8 Tipo INTEGER 778
A.2.9 Tipo BOOLEAN 778
A.2.10 Tipo ENUMERATION 778
A.2.11 Objetos de datos CONSTANT 779
A.2.12 Objetos de datos VARIABLE 779
A.2.13 Conversión de tipos 779
A.2.14 Arreglos 780
A.3 Operadores 781
A.4 Entidad de diseño de VHDL 781
A.4.1 Declaración ENTITY 782
A.4.2 Arquitectura 782
A.5 Paquete 784
A.6 Uso de subcircuitos 785
A.6.1 Declaración de un componente en un paquete 787
A.7 Instrucciones de asignación concurrente 788
A.7.1 Asignación de señal simple 789
A.7.2 Asignación de los valores de señal por medio de OTHERS 790

A.7.3 Asignación de señal seleccionada 791
A.7.4 Asignación de señal condicional 792
A.7.5 Instrucción GENERATE 793
A.8 Definición de una entidad con GENERIC 793
A.9 Instrucciones de asignación secuenciales 794
A.9.1 Instrucción PROCESS 794
A.9.2 Instrucción IF 796
A.9.3 Instrucción CASE 796
A.9.4 Instrucciones LOOP 797
A.9.5 Uso de un proceso para un circuito combinacional 797
A.9.6 Orden de las instrucciones 799
A.9.7 Uso de una variable en un proceso 800
A.10 Circuitos secuenciales 805
A.10.1 Un latch D asíncrono 805
A.10.2 Flip-flop D 806
A.10.3 Uso de una instrucción WAIT UNTIL 807
A.10.4 Un flip-flop con *reset* asíncrono 808
A.10.5 *Reset* síncrono 808
A.10.6 Registros 808
A.10.7 Registros de corrimiento 811
A.10.8 Contadores 813
A.10.9 Uso de subcircuitos con parámetros GENERIC 813
A.10.10 Una máquina de estado finito tipo Moore 816
A.10.11 Una máquina de estado finito tipo Mealy 818
A.11 Errores comunes en el código de VHDL 821
A.12 Comentarios finales 824
Bibliografía 825

Apéndice B
TUTORIAL 1 USO DEL SOFTWARE CAD QUARTUS II 827

B.1 Introducción 827
B.1.1 Primeros pasos 828
B.2 Cómo empezar un proyecto nuevo 830
B.3 Ingreso del diseño utilizando la captura esquemática 832
B.3.1 Uso del editor de bloques 832
B.3.2 La síntesis de un circuito a partir del esquema 840
B.3.3 Simulación del circuito diseñado 842
B.4 Ingreso del diseño con VHDL 846
B.4.1 Creación de otro proyecto 848
B.4.2 Uso del editor de texto 848
B.4.3 Síntesis de un circuito a partir del código de VHDL 850
B.4.4 Ejecución de la simulación funcional 850
B.4.5 Cómo usar Quartus II para corregir errores código de VHDL 850
B.5 Combinación de métodos de ingreso del diseño 851
B.5.1 Uso de un ingreso esquemático en nivel alto 851
B.5.2 Uso de VHDL en el nivel alto 854
B.6 Ventanas de Quartus II 856
B.7 Comentarios finales 858

Apéndice C
TUTORIAL 2 IMPLEMENTACIÓN DE CIRCUITOS EN DISPOSITIVOS DE ALTERA 859

C.1 Implementación de un circuito en un CPLD MAX 7000 859
C.1.1 Selección de un chip 860
C.1.2 Compilación del proyecto 861
C.1.3 Realización de la simulación de tiempo 862
C.1.4 Uso del editor de pines (Floorplan Editor) 863
C.2 Implementación de un circuito en un FPGA Cyclone 864
C.3 Implementación de un sumador con Quartus II 866
C.3.1 El código del sumador de acarreo en cascada 867
C.3.2 Simulación del circuito 868
C.3.3 Simulación de tiempo 871
C.3.4 Implementación en un chip CPLD 874
C.4 Uso de un módulo LPM 876
C.5 Diseño de una máquina de estado finito 881
C.5.1 Implementación en un CPLD 882
C.5.2 Implementación en un FPGA 886
C.6 Comentarios finales 887

Apéndice D
TUTORIAL 3 IMPLEMENTACIÓN FÍSICA EN UN PLD 889

D.1 Asignaciones de pines 889
D.1.1 Análisis de las asignaciones de pines con el editor de pines 892
D.1.2 Recompilación del proyecto con asignaciones de pines 892
D.1.3 Cómo cambiar las asignaciones de pines con Floorplan Editor 894
D.2 Descarga de un circuito en un dispositivo 895
D.3 Comentarios finales 897

Apéndice E
DISPOSITIVOS COMERCIALES 899

E.1 PLD simples 899
E.1.1 El dispositivo PAL 22V10 899
E.2 PLD complejos 901
E.2.1 MAX 7000 de Altera 902
E.3 Arreglos de compuerta programables por campo 904
E.3.1 FLEX 10K de Altera 904
E.3.2 XC4000 de Xilinx 907
E.3.3 APEX 20K de Altera 909
E.3.4 Stratix de Altera 909
E.3.5 Cyclone de Altera 911
E.3.6 Stratix II de Altera 911
E.3.7 Virtex de Xilinx 912
E.3.8 Virtex-II y Virtex-II Pro de Xilinx 912
E.3.9 Spartan-3 de Xilinx 913
E.4 Lógica de transistor a transistor 914
E.4.1 Familias de circuitos TTL 914
Bibliografía 916

RESPUESTAS 917

ÍNDICE 933

capítulo

1

Conceptos de diseño

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- Los componentes de hardware digital
- Panorama de la tecnología de los circuitos integrados
- El proceso de diseño del hardware digital

Este libro trata acerca de circuitos lógicos a partir de los que se construyen las computadoras. La comprensión cabal de este tema es indispensable para los ingenieros eléctricos y en computación de la actualidad. Los circuitos lógicos son los componentes principales de las computadoras, además de que se emplean en muchas otras aplicaciones. Se encuentran en productos de uso común, como relojes digitales, diversos aparatos electrodomésticos, reproductores de discos compactos (CD) y juegos electrónicos, así como en sistemas grandes, por ejemplo, los equipos para redes de telefonía y televisión.

A lo largo de la obra se presentan al lector los diversos temas inherentes al diseño de los circuitos lógicos. Se explican las ideas clave con ejemplos simples y se muestra cómo derivar circuitos complejos a partir de los elementales. La teoría clásica aplicada en el diseño de los circuitos lógicos se cubre con gran profundidad a fin de ofrecer al lector una comprensión intuitiva de la naturaleza de tales circuitos. Por añadidura, en cada capítulo se ilustra la forma actual de diseñar circuitos lógicos por medio de modernas herramientas de software de *diseño asistido por computadora* (CAD, *computer aided design*). El método CAD adoptado en la obra se basa en VHDL, el lenguaje de diseño estándar en la industria. En el capítulo 2 se explica el diseño con VHDL, y su empleo junto con las herramientas CAD es parte integral de cada capítulo.

Los circuitos lógicos se implementan electrónicamente mediante transistores en un chip de circuito integrado. La tecnología moderna permite fabricar chips que contienen decenas de millones de transistores, como en el caso de los procesadores de computadora. Los bloques básicos de tales circuitos son fáciles de entender, pero no hay nada simple en torno a un circuito que contiene decenas de millones de transistores. La complejidad que conlleva el gran tamaño de los circuitos lógicos puede manejarse bien sólo si se aplican técnicas de diseño sumamente organizadas. En el presente capítulo se estudian estas técnicas, pero primero se describe de modo breve la tecnología de hardware utilizada para construir circuitos lógicos.

1.1 HARDWARE DIGITAL

Los circuitos lógicos se usan para construir hardware de computadora, así como otros tipos de productos, que se clasifican en términos generales como *hardware digital*. La razón del nombre *digital* será clara más adelante en el libro: procede de la forma en la que se representa la información en las computadoras: como señales electrónicas que corresponden a dígitos de información.

La tecnología empleada para construir hardware digital evolucionó en forma sorprendente durante las últimas cuatro décadas. Hasta el decenio de 1960 los circuitos lógicos se construían con componentes voluminosos, como transistores y resistores que venían como partes individuales. El advenimiento de los circuitos integrados hizo posible colocar varios transistores y, por tanto, un circuito entero en un solo chip. Aunque al principio estos circuitos sólo tenían unos cuantos transistores, conforme la tecnología mejoró se volvieron más grandes. Los chips de circuitos integrados se fabrican sobre una oblea de silicio como la que se muestra en la figura 1.1. La oblea se corta para producir los chips individuales, que luego se colocan en el interior de un tipo especial de paquete de chip. Hacia 1970 fue posible implementar todos los circuitos necesarios para elaborar un microprocesador en un solo chip. Aunque los primeros microprocesadores tenían modestas capacidades computacionales para los estándares de la actualidad, abrieron la puerta a la revolución del procesamiento al proporcionar los medios para la construcción de computadoras personales asequibles a la gente común. Hace aproximadamente 30 años Gordon Moore, gerente de Intel Corporation, observó que la tecnología de los circuitos integrados progresaba a un ritmo sorprendente y cada 1.5 a 2 años duplicaba el número de transistores que podían colocarse en un chip. Este fenómeno, conocido de manera informal como *ley de Moore*, aún se presenta hoy en día. Por ende, si a principios del decenio de 1990 los microprocesadores

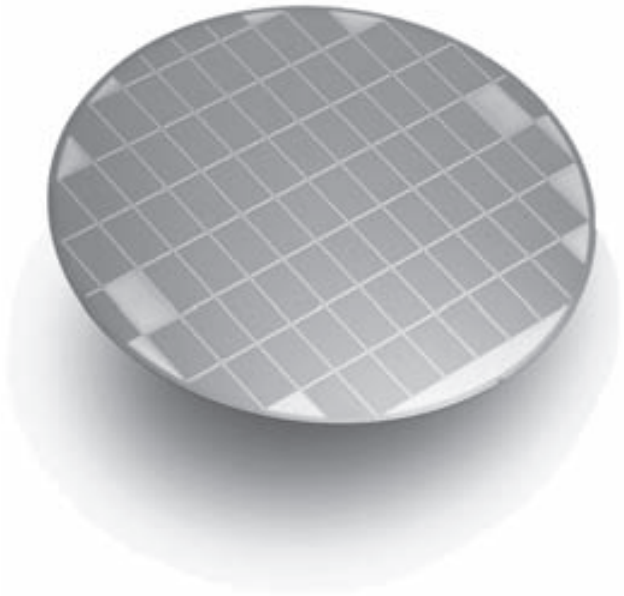


Figura 1.1 Oblea de silicio (cortesía de Altera Corp.).

podían fabricarse con unos cuantos millones de transistores, hacia finales de ese decenio fue posible manufacturar chips que contenían más de 10 millones de transistores. Los chips actuales pueden tener algunos cientos de millones de transistores.

Cabe esperar que la ley de Moore se cumpla durante por lo menos los 10 años siguientes. La SIA (Semiconductor Industry Association), un consorcio de fabricantes de circuitos integrados, publica una estimación de cómo espera que evolucione la tecnología. Conocida como la *Guía SIA* [1], en ella se predice el tamaño mínimo de un transistor que puede fabricarse en un chip de circuito integrado. El tamaño de un transistor se mide con un parámetro denominado *longitud de compuerta*, que se analiza en el capítulo 3. La tabla 1.1 muestra un ejemplo de la *Guía SIA*. En el 2004, la mínima longitud de compuerta posible que podía fabricarse con confiabilidad era de 90 nm. En la primera fila de la tabla se indica que para el 2012 se espera que la longitud de

Tabla 1.1 Muestra de la *Guía SIA*.

	Año					
	1999	2001	2004	2006	2009	2012
Longitud de compuerta de transistor	0.14 μm	0.12 μm	90 nm	65 nm	50 nm	35 nm
Transistores por cm^2	14 millones	16 millones	24 millones	40 millones	64 millones	100 millones
Tamaño de chip	800 mm^2	850 mm^2	900 mm^2	1 000 mm^2	1 100 mm^2	1 300 mm^2

compuerta mínima se reduzca de manera constante hasta alrededor de 35 nm. El tamaño de un transistor determina cuántos de ellos pueden colocarse en cierta área del chip; en la actualidad esa cifra alcanza un máximo cercano a 30 millones de transistores por centímetro cuadrado. Se estima que este número alcanzará los 100 millones para el 2012. Se considera que el tamaño del chip más grande será de aproximadamente 1 300 mm^2 para ese entonces; por ende, ¡podría haber chips hasta con mil trescientos millones de transistores! No hay duda de que esta tecnología tendrá un gran impacto en todos los aspectos de la vida de las personas.

El diseñador de hardware digital podría enfrentar el diseño de circuitos lógicos que puedan implementarse en un solo chip o, más probablemente, el diseño de circuitos que incluyan varios chips colocados sobre una *tarjeta de circuito impreso* (PCB, *printed circuit board*). Con frecuencia, parte de los circuitos lógicos puede hallarse en chips que son fácilmente adquiribles. Esta situación simplifica la tarea de diseño y acorta el tiempo necesario para desarrollar el producto final. Antes de explicar el proceso de diseño con más detalle es preciso presentar los diferentes tipos de chips de circuitos integrados que pueden usarse.

Una gran variedad de chips cumple diversas funciones útiles en el diseño de hardware digital. Van desde los más simples, con poca funcionalidad, hasta los extremadamente complejos. Por ejemplo, un producto de hardware digital puede requerir un microprocesador para realizar ciertas operaciones aritméticas, chips de memoria que provean capacidad de almacenamiento y chips de conexión que permitan enlazar sin problema los dispositivos de entrada y salida. Tales chips están disponibles en el mercado con diversos proveedores.

Para el grueso de los productos de hardware digital también es necesario diseñar y construir algunos circuitos lógicos desde el principio. Para implementar estos circuitos es posible usar tres tipos principales de chips: los chips estándar, los dispositivos lógicos programables y los chips "a la medida". Todos ellos se analizan a continuación.

1.1.1 CHIPS ESTÁNDAR

Ya hay muchos chips que forman algunos circuitos lógicos usados comúnmente; nos referiremos a ellos como *chips estándar*, ya que cumplen un estándar con el que se está de acuerdo en términos de funcionalidad y configuración física. Cada chip estándar contiene una pequeña cantidad de circuitos (casi siempre menos de 100 transistores) y efectúa una función simple. Para construir un circuito lógico el diseñador elige los chips que llevan a cabo las funciones necesarias y luego define cómo deben interconectarse para formar un circuito lógico más grande.

Los chips estándar fueron populares para construir circuitos lógicos hasta principios de la década de 1980. Sin embargo, a medida que la tecnología de circuitos integrados mejoró, se volvió ineficiente usar espacio valioso en las PCB para chips con menor funcionalidad. Otro inconveniente de los chips estándar es que su funcionalidad es fija y no puede cambiarse.

1.1.2 DISPOSITIVOS LÓGICOS PROGRAMABLES

En contraste con los chips estándar, cuyas funciones son fijas, es posible construir chips con circuitos que el usuario puede configurar para usarlos en una amplia variedad de circuitos lógicos. Estos chips tienen una estructura muy general e incluyen una serie de *interruptores programables* que permiten configurar sus circuitos internos en muchas formas. El diseñador puede

Figura 1.2 Chip con arreglo de compuerta de campos programables (cortesía de Altera Corp.).

implementar cualesquiera funciones que necesite para una aplicación particular eligiendo la configuración apropiada de los interruptores. El usuario final programa los interruptores, en lugar de que se configuren cuando se fabrica el chip. Este tipo de chips se conoce como *dispositivos lógicos programables* (PLD, *programmable logic devices*) y se estudian en el capítulo 3.

La mayor parte de los PLD puede programarse en repetidas ocasiones, lo que constituye una ventaja, pues un diseñador que desarrolle el prototipo de un producto puede programar un PLD para cumplir cierta función y, más tarde, cuando el hardware prototipo se ponga a prueba, efectuar correcciones mediante la reprogramación del PLD. La reprogramación puede ser necesaria, por ejemplo, si el diseño de una función no está muy apegado a lo que se quería o si se precisan funciones nuevas que no se consideraron en el diseño original.

Los PLD están disponibles en diversos tamaños. Sirven para formar circuitos lógicos mucho más grandes de lo que un chip estándar puede realizar. Gracias a su tamaño y a que pueden ajustarse para satisfacer los requisitos de una aplicación específica, se usan mucho en la actualidad. Uno de los tipos de PLD más modernos se conoce como *arreglo de compuertas de campos programables* (FPGA, *field-programmable gate array*). Ahora existen FPGA que contienen más de 500 millones de transistores [2, 3]. En la figura 1.2 se muestra la fotografía de un chip FPGA, el cual consta de un gran número de pequeños elementos de circuito lógico que pueden conectarse en conjunto mediante los interruptores programables. Los elementos del circuito lógico se ordenan en una estructura bidimensional regular.

1.1.3 CHIPS DISEÑADOS A LA MEDIDA

Los PLD están disponibles como componentes comerciales que pueden adquirirse con diferentes proveedores. Como son programables, se usan para implementar la mayoría de los circuitos lógicos que se encuentran en el hardware digital. Sin embargo, tienen el inconveniente de que los interruptores programables ocupan una buena parte del área del chip y limitan la rapidez de operación de los circuitos implementados. Por ende, en algunos casos los PLD no satisfacen los objetivos de desempeño o costo deseados. En tales situaciones es posible diseñar un chip desde

cero; primero se diseñan los circuitos lógicos que deben incluirse en el chip y luego se elige una tecnología apropiada para implementarlo. Por último, el chip lo fabrica una compañía que tiene las instalaciones para su elaboración. Este enfoque se conoce como *diseño a la medida* o *casi a la medida* y tales chips se denominan *chips a la medida* o *casi a la medida*. La intención es que esos chips se utilicen en aplicaciones específicas, por lo que a veces reciben el nombre de *circuitos integrados específicos para una aplicación* (ASIC, *application-specific integrated circuits*).

La ventaja principal de un chip a la medida es que es posible optimar su diseño para que cumpla una tarea concreta, lo que suele desembocar en un mejor desempeño. Puede incluirse una cantidad más grande de circuitos lógicos en un chip a la medida de lo que sería posible en otros tipos de chips. El costo de producir tales chips es elevado, pero si se utilizan en un producto que se venda en grandes cantidades, entonces el costo por chip, amortizado sobre el número total de chips fabricados, tal vez sea menor que el costo total de los chips comerciales que se necesitarían para implementar la misma función. Más aún, si puede emplearse un solo chip en vez de muchos de ellos para lograr la misma meta, entonces se requiere un área menor en una PCB que albergue los chips en el producto final. Ello resulta en una ulterior reducción del costo.

Una desventaja del enfoque del diseño a la medida es que la fabricación de un chip suele consumir una cantidad considerable de tiempo, del orden de meses. En contraste, si en su lugar es posible usar un PLD, entonces el usuario final puede programar los chips y no hay demoras en la fabricación.

1.2 EL PROCESO DE DISEÑO

La disponibilidad de herramientas basadas en computadoras influyó enormemente en el proceso de diseño en numerosos entornos. Por ejemplo, el enfoque global del diseño de un automóvil es similar al de un mueble o una computadora. Hay que seguir ciertos pasos en el ciclo de desarrollo si el producto final ha de satisfacer los objetivos establecidos. A continuación se presenta un ciclo de desarrollo típico en los términos más generales; luego nos centraremos en los aspectos particulares propios del diseño de circuitos lógicos.

El diagrama de flujo de la figura 1.3 bosqueja un proceso de desarrollo típico. Supóngase que se trata de desarrollar un producto que satisfaga ciertas expectativas. Los requisitos más obvios son que el producto tiene que funcionar de manera adecuada, ha de satisfacer un nivel de desempeño esperado y su costo no debe exceder cierto límite.

El proceso empieza con la definición de las especificaciones del producto. Se identifican sus características esenciales y se establece un método aceptable para evaluarlas una vez implementadas en el producto final. Las especificaciones han de ser lo suficientemente concretas para garantizar que el producto desarrollado satisfará las expectativas generales, pero no innecesariamente restrictivas (es decir, no deben impedir opciones de diseño que puedan conducir a ventajas imprevistas).

A partir de un conjunto completo de especificaciones es necesario definir la estructura general de un diseño inicial del producto. Este paso es difícil de automatizar. Suele realizarlo un diseñador humano porque no existe una estrategia obvia para desarrollar la estructura global de un producto: se requiere considerable experiencia e intuición en el diseño.

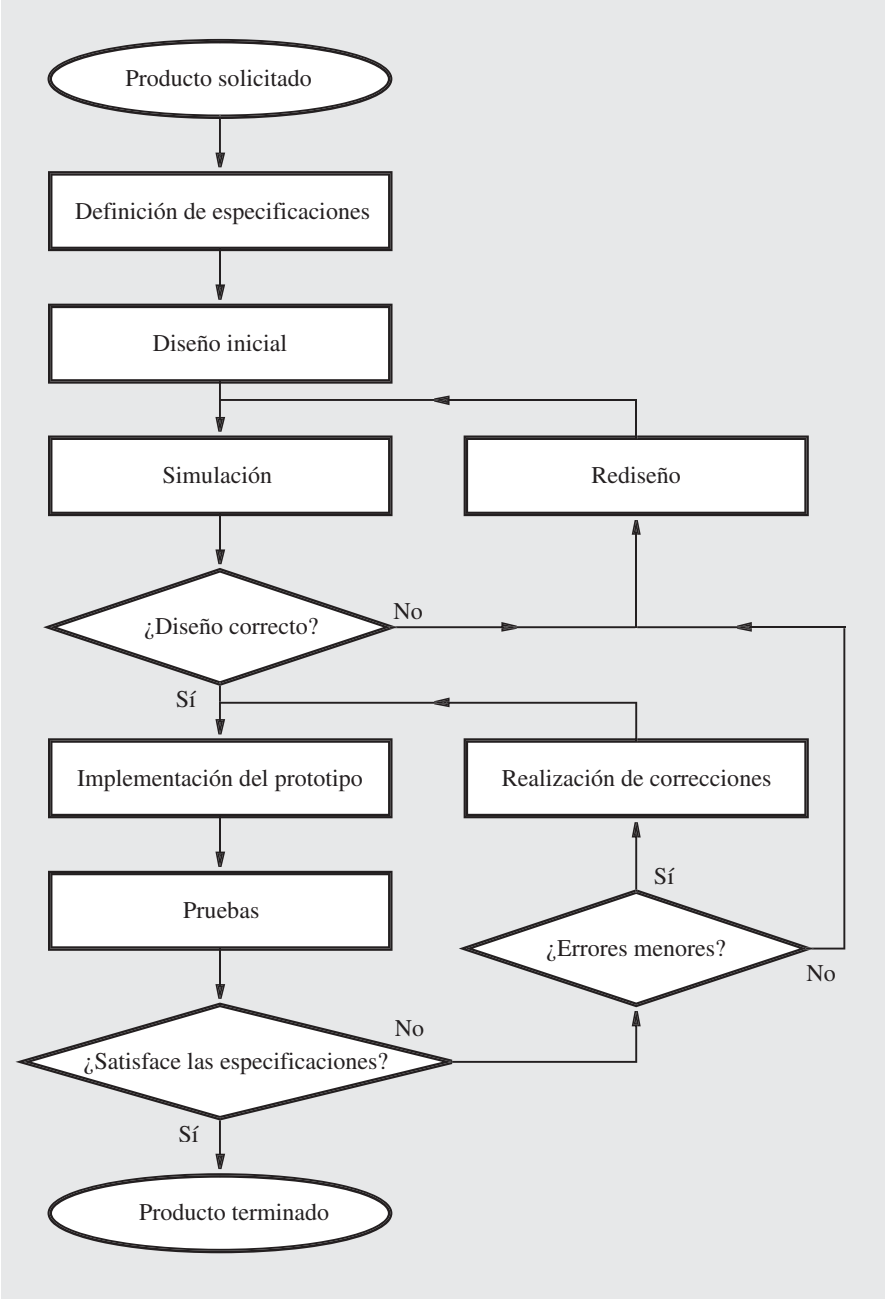


Figura 1.3 El proceso de desarrollo.

Después de establecer la estructura general, se usan herramientas CAD para afinar los detalles. Hay muchos tipos de herramientas CAD, que van desde las que ayudan con el diseño de las partes individuales del sistema hasta las que permiten representar en una computadora la estructura global del producto completo. Una vez concluido el diseño inicial los resultados se confrontan con las especificaciones originales. Tradicionalmente, antes de la llegada de las herramientas CAD, este paso implicaba la construcción de un modelo físico del producto diseñado, que por lo general incluía sólo las partes principales. Hoy en día rara vez es necesario hacer eso. Las herramientas CAD permiten a los diseñadores simular el comportamiento de productos increíblemente complejos y tales simulaciones se usan para determinar si el diseño obtenido satisface las especificaciones requeridas. Si se encuentran errores, entonces se realizan los cambios adecuados y se repite la verificación del nuevo diseño mediante simulación. Aunque algunos errores de diseño pueden escapar de la detección mediante la simulación, casi todos los problemas se descubren de esta forma, salvo los más sutiles.

Cuando la simulación indica que el diseño es correcto se construye un prototipo físico completo del producto. El prototipo se pone a prueba de manera rigurosa para comprobar su conformidad con las especificaciones. Cualesquiera errores revelados en las pruebas han de corregirse. Los errores pueden ser menores y con frecuencia es posible eliminarlos con pequeñas enmiendas directas en el prototipo del producto. En caso de grandes errores es preciso rediseñar el producto y repetir los pasos antes explicados. Cuando el prototipo pasa todas las pruebas, el producto se juzga bien diseñado y puede irse a producción.

1.3 DISEÑO DE HARDWARE DIGITAL

La explicación previa del proceso de desarrollo es relevante en términos generales. Los pasos esbozados en la figura 1.3 se aplican por completo en el desarrollo de hardware digital. Antes de analizar toda la secuencia de pasos en este entorno de desarrollo hay que hacer hincapié en la naturaleza iterativa del proceso de diseño.

1.3.1 CICLO DE DISEÑO BÁSICO

Cualquier proceso de diseño comprende una secuencia básica de tareas (figura 1.4) que se efectúan en varias situaciones. Si se supone que se tiene un concepto primario acerca de lo que hay que lograr en el proceso de diseño, el primer paso consiste en generar un diseño inicial. Este paso requiere mucho esfuerzo manual porque el grueso de los diseños tiene ciertas metas específicas que sólo se alcanzan por medio del conocimiento, las habilidades y la intuición del diseñador. El siguiente paso es la simulación del diseño a mano. Se dispone de excelentes herramientas CAD para auxiliar en esta etapa. Para que la simulación tenga éxito es preciso tener adecuadas condiciones de entrada que puedan aplicarse al diseño que se simula y más tarde al producto final que se someterá a pruebas. Al aplicar estas condiciones de entrada el simulador intenta comprobar que el producto diseñado se desempeñará como se requiere según las especificaciones del producto original. Si la simulación revela algunos errores hay que cambiar el diseño a fin de superarlos. La versión rediseñada se simula de nuevo para determinar si los errores desaparecieron. Este ciclo se repite hasta que la simulación indica un buen diseño. Un diseñador prudente dedica esfuerzos considerables a remediar los errores durante la simulación porque éstos suelen ser

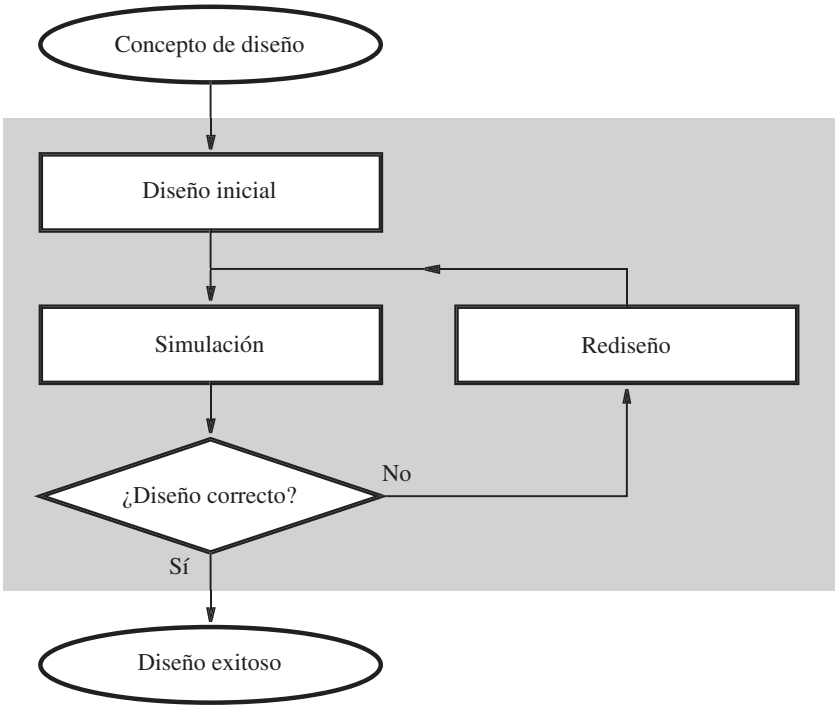


Figura 1.4 Ciclo de diseño básico.

mucho más difíciles de corregir si se descubren tarde en el proceso de diseño. Aun así, algunos de ellos no se detectan durante la simulación, por lo que hay que enfrentarlos en etapas posteriores del ciclo de desarrollo.

1.3.2 ESTRUCTURA DE UNA COMPUTADORA

Para entender la función de los circuitos lógicos en los sistemas digitales considérese la estructura de una computadora típica como la que se ilustra en la figura 1.5*a*. El gabinete contiene varias tarjetas de circuito impreso (PCB), una fuente de poder y unidades de almacenamiento (no se muestran en la figura), como un disco duro y unidades de DVD o CD-ROM. Todas las unidades se conectan a una PCB principal, llamada *tarjeta madre*. Como se indica en la parte inferior de la figura, la tarjeta madre contiene varios chips de circuitos integrados y provee ranuras para conectar otras PCB, como tarjetas de audio, video y red.

En la figura 1.5*b* se observa la estructura de un chip de circuito integrado, el cual comprende varios subcircuitos, que se interconectan para construir el circuito completo. Ejemplos de subcircuitos son los que realizan operaciones aritméticas, almacenan datos o controlan el flujo de éstos. Cada uno de dichos subcircuitos es un circuito lógico. Como se muestra a mitad de la figura,

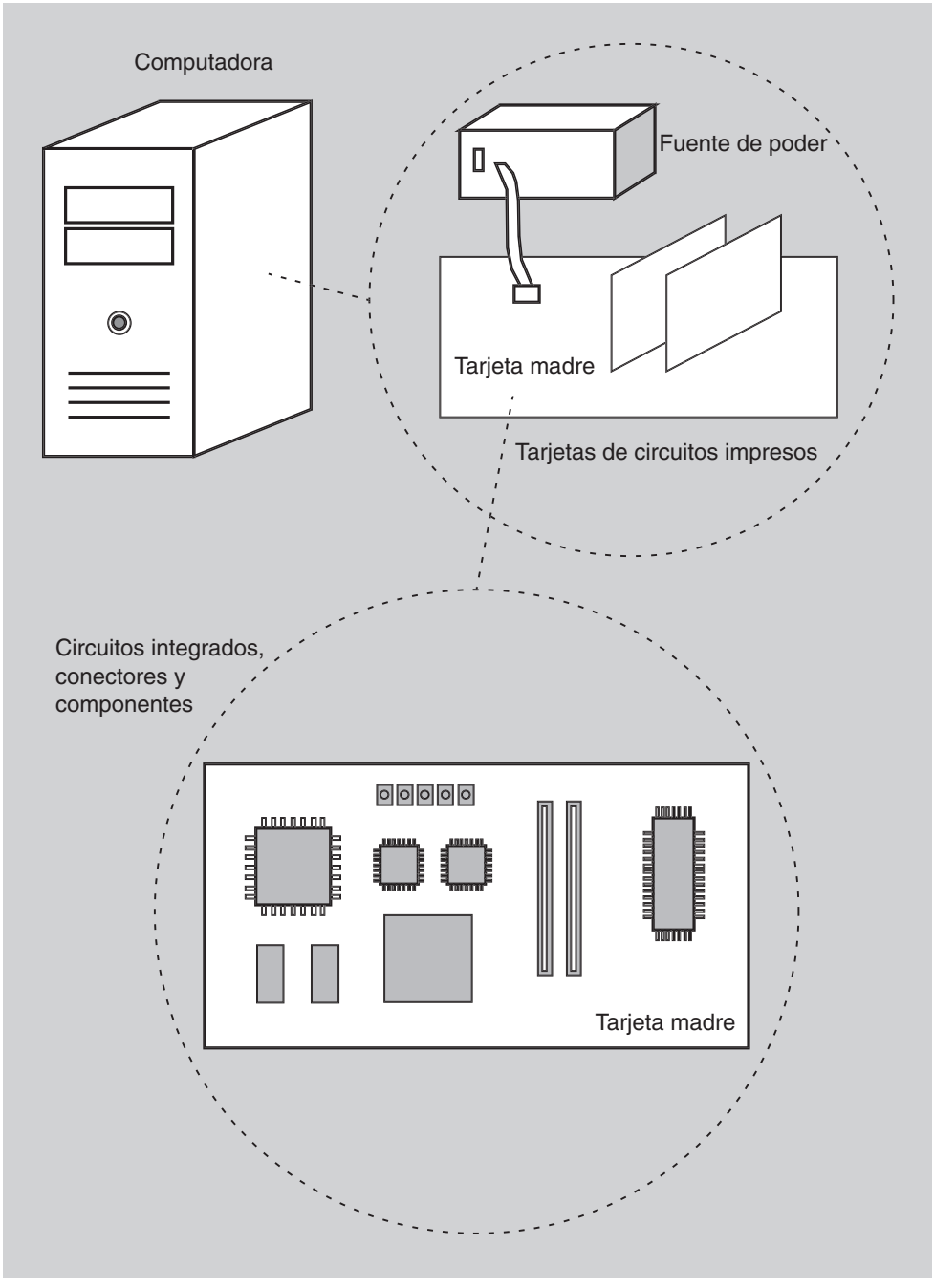


Figura 1.5 Sistema de hardware digital (parte *a*).

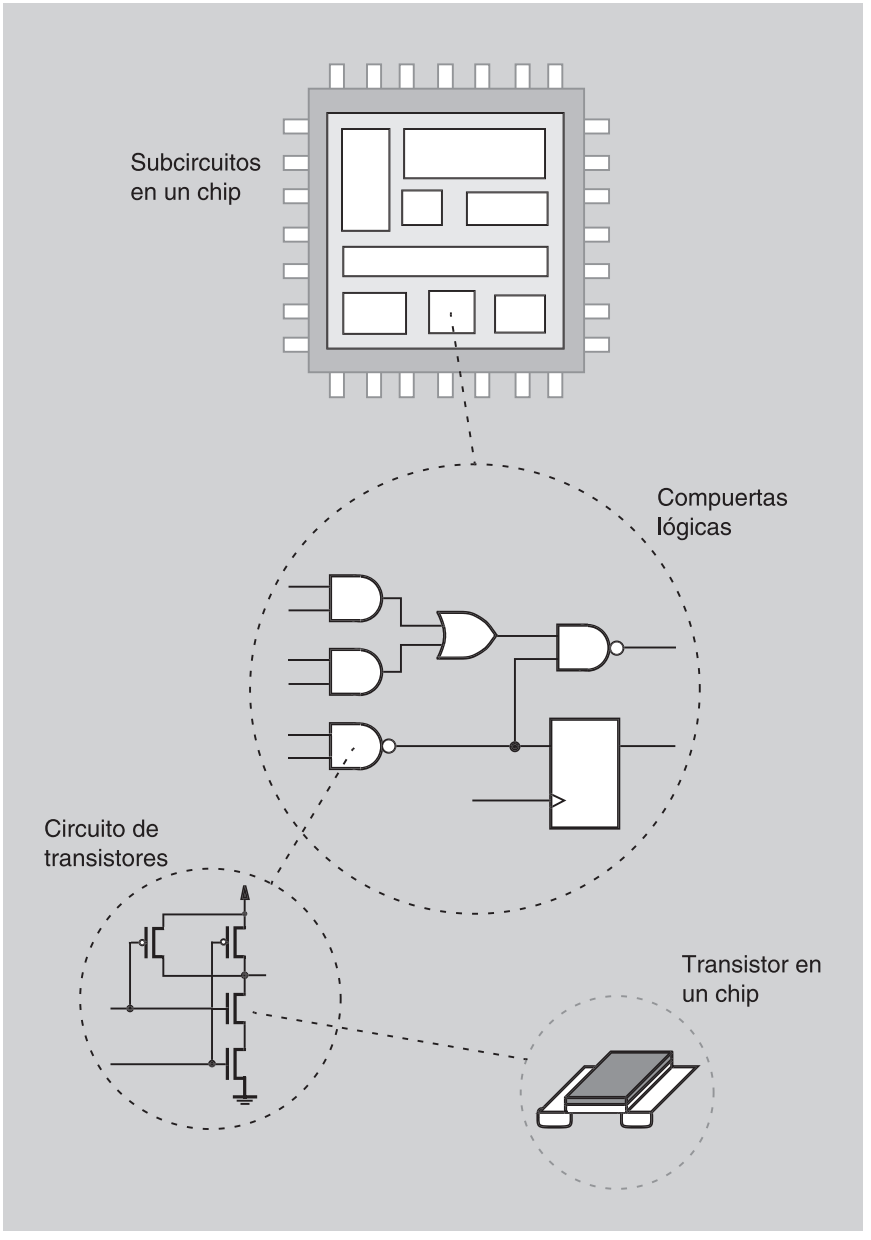


Figura 1.5 Sistema de hardware digital (parte *b*).

un circuito lógico comprende una red de *compuertas lógicas* conectadas. Cada compuerta lógica realiza una función muy simple, y las operaciones más complejas las efectúan las compuertas conectadas en conjunto. Las compuertas lógicas se construyen con transistores, que a su vez se implementan mediante la fabricación de varias capas de material sobre un chip de silicio.

Esta obra se ocupa principalmente de la parte central de la figura 1.5*b*: el diseño de circuitos lógicos. Se explica cómo diseñar circuitos que desempeñan funciones importantes, como sumar, restar o multiplicar números, llevar conteos, almacenar datos y controlar el procesamiento de la información. También se muestra la forma de especificar el comportamiento de tales circuitos, cómo se diseñan para lograr costos mínimos o máxima rapidez de operación, y la manera en que se prueban para garantizar el funcionamiento correcto. Asimismo se explica brevemente cómo operan los transistores y cómo se construyen sobre los chips de silicio.

1.3.3 DISEÑO DE UNA UNIDAD DE HARDWARE DIGITAL

Como se muestra en la figura 1.5, los productos de hardware digital llevan una o más PCB que contienen muchos chips y otros componentes. El desarrollo de tales productos comienza con la definición de la estructura global. Luego se eligen los chips de circuitos integrados que se requieren y se diseña la PCB que los alberga y conecta. Si los chips seleccionados incluyen PLD o chips a la medida, entonces estos chips deben diseñarse antes de emprender el diseño en el nivel de la PCB. Puesto que la complejidad de los circuitos implementados en chips individuales y en las tarjetas de circuito es muy elevada, resulta esencial utilizar buenas herramientas CAD.

En la figura 1.6 se muestra una fotografía de una PCB. La PCB es parte de un gran sistema de cómputo diseñado en la Universidad de Toronto. Esta computadora, llamada *NUMAchine* [4, 5], es un *multiprocesador*, lo que significa que contiene muchos procesadores que pueden usarse juntos para encarar una tarea en particular. La PCB de la figura contiene un chip procesador y varios chips de memoria y apoyo. Se necesitan complejos circuitos lógicos para interconectar el procesador y el resto del sistema. Para implementar tales circuitos lógicos se usan varios PLD.

A fin de ilustrar el ciclo de desarrollo completo con más detalle, se considerarán los pasos necesarios para producir una unidad de hardware digital que puede implementarse sobre una PCB. Este hardware podría verse como un circuito lógico muy complejo que realiza las funciones definidas en las especificaciones del producto. En la figura 1.7 se muestra el flujo de diseño, si se supone que se tiene un concepto de diseño que define el comportamiento esperado y las características de este gran circuito.

Una forma ordenada de lidiar con la complejidad inherente es dividir el circuito en bloques más pequeños y luego diseñar cada uno de ellos por separado. El enfoque consistente en la división de una gran tarea en partes más pequeñas y manejables recibe el nombre de *enfoque divide y vencerás*. El diseño de cada bloque sigue el procedimiento descrito en la figura 1.4. Se definen los circuitos en cada bloque y luego se eligen los chips necesarios para implementarlos. Se simula la operación de estos circuitos y se hacen las correcciones necesarias.

Una vez que se tiene el diseño correcto de todos los bloques, se define su interconexión, con la que efectivamente se les combina en un solo bloque. Ahora es necesario simular este circuito completo y corregir los errores. Según los errores hallados, puede ser necesario regresar a pasos previos, como indican las trayectorias *A*, *B* y *C* del diagrama de flujo. Es factible que algunos errores se deban a conexiones incorrectas entre los bloques, por lo que habrá que redefinirlas siguiendo la trayectoria *C*. Tal vez algunos bloques no se diseñaron correctamente, por lo que

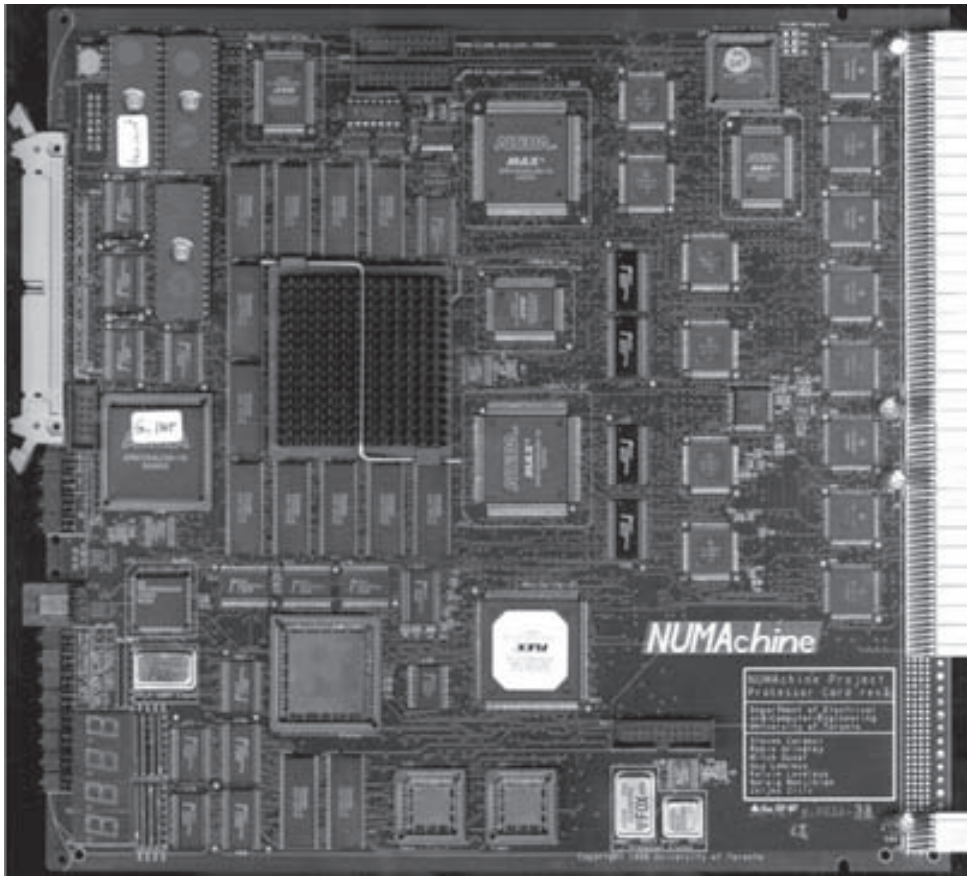


Figura 1.6 Tarjeta de circuito impreso.

habrá que seguir la trayectoria *B* y rediseñar los bloques erróneos. Otra posibilidad es que el mismísimo primer paso de dividir en bloques el gran circuito global no se realizara bien, en cuyo caso se sigue la trayectoria *A*. Esto puede ocurrir, por ejemplo, si ninguno de los bloques implementa cierta funcionalidad necesaria en el circuito completo.

La conclusión correcta de la simulación funcional indica que el circuito diseñado cumplirá bien todas sus funciones. El siguiente paso es decidir cómo materializar este circuito en una PCB. Hay que determinar la ubicación física de cada chip en la tarjeta, así como definir el esquema de cableado necesario para conectar los chips. Este paso recibe el nombre de *diseño físico* de la PCB. Para llevarlo a cabo automáticamente se recibe un enorme apoyo de las herramientas CAD.

Una vez establecidas tanto la ubicación de los chips como las conexiones de cables en la PCB, es deseable ver cómo este esquema físico afectará el desempeño del circuito en la tarjeta terminada. Resulta razonable suponer que si la simulación funcional previa indicó que todas las funciones se realizarán correctamente, entonces las herramientas CAD usadas en el paso de diseño físico garantizarán que el comportamiento funcional requerido no se corromperá con la colocación de los chips en la tarjeta ni con su conexión mediante los cables para formar el

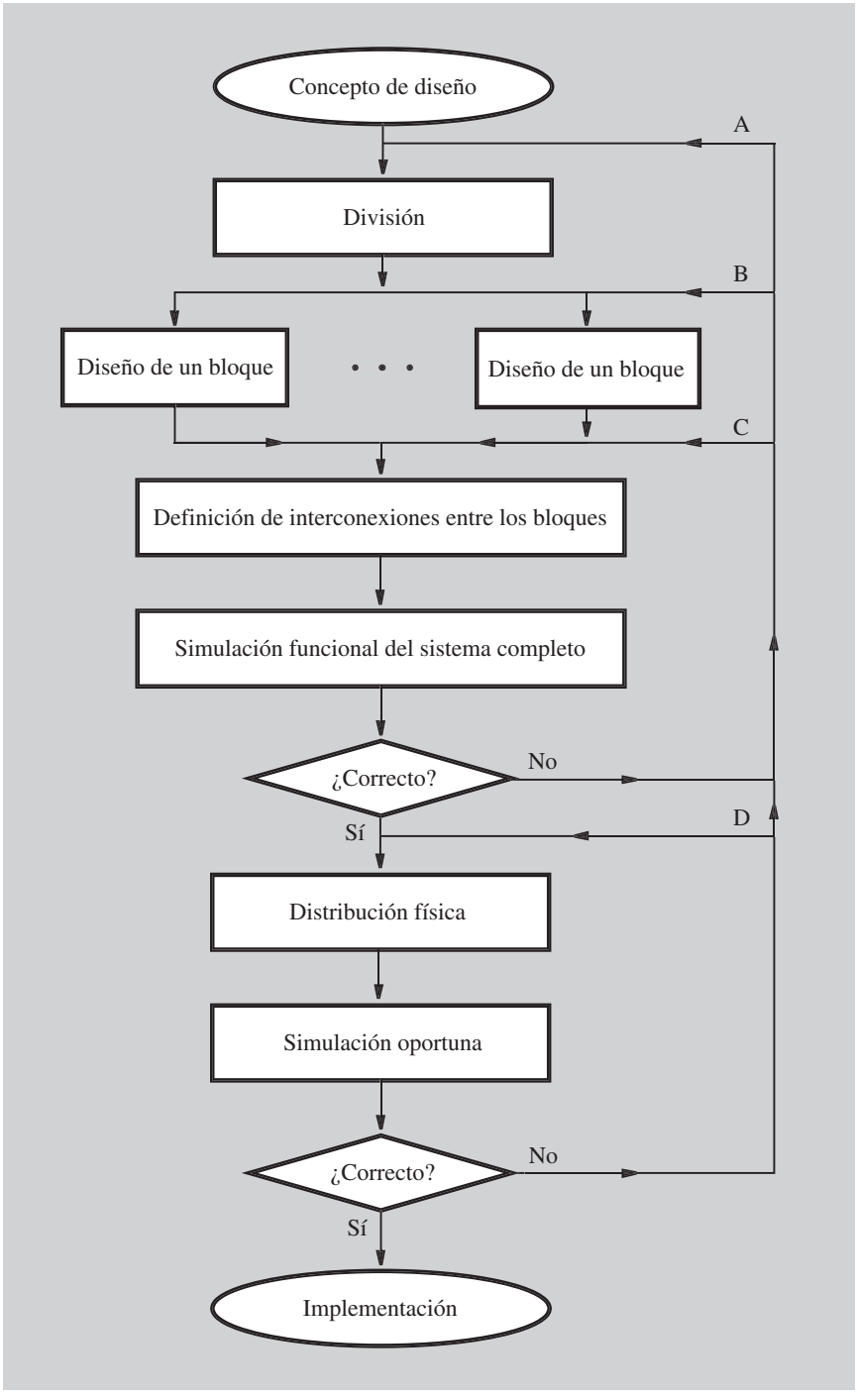


Figura 1.7 Flujo de diseño para circuitos lógicos.

circuito final. Sin embargo, aun cuando el comportamiento funcional sea correcto, el circuito podría operar más lentamente de lo deseado y, por tanto, conducir a un desempeño inadecuado. Esta deficiencia se presenta porque el cableado físico en la PCB implica trazas metálicas que presentan resistencia y capacitancia a las señales eléctricas, por lo que pueden tener un efecto significativo en la rapidez de operación. Para distinguir entre la simulación que sólo considera la funcionalidad del circuito y la que también tiene en cuenta el comportamiento oportuno se emplean los términos *simulación funcional* y *simulación oportuna*. Una simulación oportuna puede revelar problemas potenciales de desempeño, que luego pueden corregirse mediante las herramientas CAD para realizar cambios en el diseño físico de la PCB.

Tras completar el proceso de diseño el circuito diseñado está listo para la implementación física. En la figura 1.8 se indican los pasos necesarios para implementar una tarjeta prototipo. Se construye y prueba una primera versión de la tarjeta. La mayor parte de los errores menores que se detectan puede corregirse haciendo cambios directamente en la tarjeta prototipo, lo que quizás implique cambios en el cableado o quizá la reprogramación de ciertos PLD. Los problemas más grandes precisan un rediseño más sustancial. Según la naturaleza del problema, es posible que el diseñador deba regresar a alguno de los puntos *A*, *B*, *C* o *D* en el proceso de diseño de la figura 1.7.

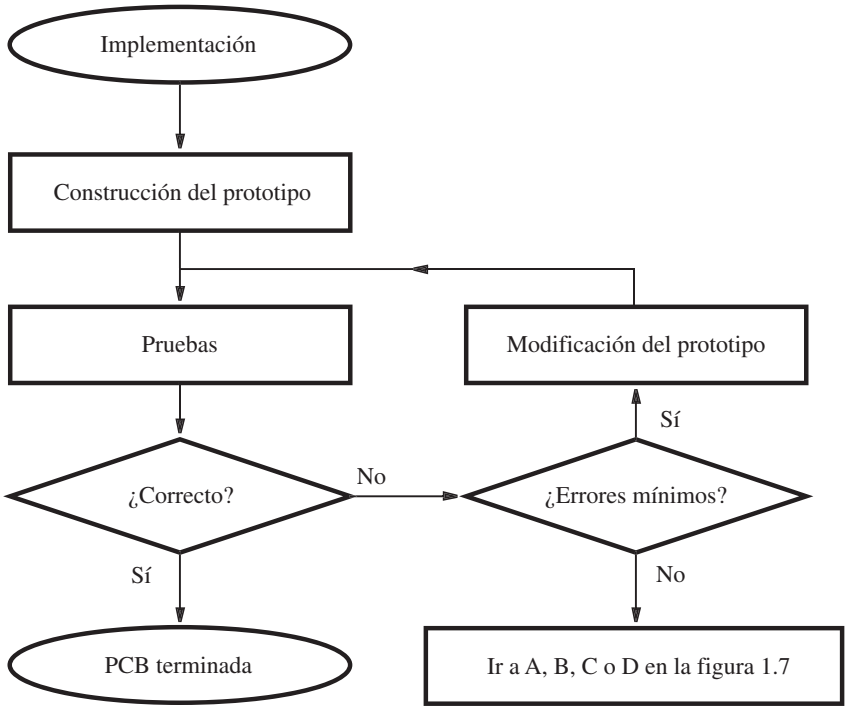


Figura 1.8 Conclusión del desarrollo de una PCB.

Hemos descrito el proceso de desarrollo en el que el circuito final se implementa usando muchos chips sobre una PCB. El material expuesto en el libro se aplica de manera directa a este tipo de problema de diseño. Sin embargo, por razones prácticas, los ejemplos de diseño presentados son relativamente pequeños y pueden materializarse en un solo circuito integrado, ya sea un chip diseñado a la medida o un PLD. Todos los pasos de la figura 1.7 también son relevantes en este caso, si se tiene en cuenta que los bloques de circuito que se diseñarán tienen una escala más pequeña.

1.4 DISEÑO DE CIRCUITOS LÓGICOS EN ESTE LIBRO

En la presente obra se usan extensivamente los PLD para ilustrar muchos aspectos del diseño de circuitos lógicos. Se seleccionó esta tecnología porque se utiliza mucho en los productos de hardware digital reales y porque el usuario puede programar los chips. La tecnología PLD es en particular adecuada para propósitos educativos, ya que muchos lectores tienen acceso a instalaciones para programar PLD, lo que les permite implementar realmente los circuitos muestra. Para ilustrar aspectos de diseño práctico, en la obra se emplean dos tipos de PLD, que son los dos tipos de dispositivos más usados en los productos actuales de hardware digital. Uno de ellos se conoce como *dispositivos lógicos programables complejos* (CPLD, *complex programmable logic devices*) y el otro como *arreglo de compuertas de campos programables* (FPGA, *field-programmable gate array*). Estos chips se estudian en el capítulo 3.

Para obtener experiencia práctica y comprender mejor los circuitos lógicos se aconseja al lector implementar los ejemplos de este libro con herramientas CAD. La mayor parte de los grandes proveedores de sistemas CAD ofrece sus herramientas mediante programas universitarios para uso educativo. Algunos ejemplos son Altera, Cadence, Mentor Graphics, Synopsys, Synplicity y Xilinx. Con esta obra se pueden usar igualmente bien los sistemas CAD que cualquiera de estas compañías ofrezca. Para quienes todavía no tienen acceso a las herramientas CAD, el disco compacto adjunto a la obra incluye el sistema Quartus II CAD de Altera. Este software de actualidad soporta todas las fases del ciclo de diseño, es poderoso, sencillo de usar y se instala fácilmente en una computadora personal. En los apéndices B, C y D se presenta una secuencia de tutoriales paso a paso para ilustrar el uso de las herramientas CAD en conjunto con el libro.

Con fines educativos, algunos fabricantes de PLD ofrecen tarjetas de laboratorio para el desarrollo de circuitos impresos que incluyen uno o más chips PLD y una interfaz a una computadora personal. Cuando se diseña un circuito lógico con las herramientas CAD, el circuito se puede descargar al PLD de la tarjeta. Luego pueden aplicarse entradas al PLD mediante simples interruptores, y examinarse las salidas generadas. Dichas tarjetas de laboratorio se describen en las páginas en Internet de los proveedores de PLD.

1.5 TEORÍA Y PRÁCTICA

El diseño moderno de circuitos lógicos depende enormemente de las herramientas CAD, pero la disciplina del diseño lógico evolucionó mucho antes que las herramientas CAD se inventaran. Esta cronología es muy obvia debido a que las primeras computadoras se construyeron con circuitos lógicos y, para ser honestos, no había computadoras en las que se pudiera diseñarlas.

Para estudiar los circuitos lógicos se han creado varias técnicas de diseño manual. El álgebra booleana, que se aborda en el capítulo 2, se adoptó como un medio matemático para representarlos. También se construyó una gran cantidad de "teoría", que muestra cómo tratar ciertos aspectos del diseño. Para tener éxito, el diseñador debe aplicar este conocimiento en la práctica.

Las herramientas CAD no sólo permiten diseñar circuitos complejos si no también simplifican el trabajo de diseño. Realizan muchas tareas de manera automática, lo que sugiere que los diseñadores actuales no necesitan comprender los conceptos teóricos aplicados en las tareas que las herramientas CAD llevan a cabo. Entonces una pregunta obvia sería: ¿por qué uno tiene que estudiar la teoría que ya no se necesita para el diseño manual? ¿Por qué no simplemente se aprende a usar las herramientas CAD?

Hay tres grandes razones para aprender la teoría. Primera, aunque las herramientas CAD desempeñan las tareas automáticas de optimación de un circuito lógico para satisfacer objetivos de diseño en particular, el diseñador debe dar la descripción original del circuito lógico. Si especifica un circuito que inherentemente tiene propiedades incorrectas, el circuito final también será de mala calidad. Segunda, las reglas y teoremas algebraicos para el diseño y el manejo de circuitos lógicos se implementan directamente en las herramientas CAD actuales. No es posible que un usuario de tales herramientas comprenda lo que hacen si no tiene la teoría implícita en ello. Tercera, las herramientas CAD ofrecen muchos pasos de procesamiento adicionales a los que un usuario puede recurrir cuando trabaja en un diseño. El diseñador elige qué opciones usar al examinar el circuito resultante que producen las herramientas CAD y decide si satisfacen los objetivos requeridos. La única forma en que puede saber si aplica o no una opción en cierta situación es saber qué harán las herramientas CAD si invoca esa opción; de nuevo, esto implica que el diseñador debe conocer la teoría implícita. En este libro se estudiará la teoría de circuitos lógicos clásica porque no es posible convertirse en un buen diseñador de circuitos lógicos sin comprender los conceptos fundamentales.

Como nota final, hay otra buena razón para aprender parte de la teoría de los circuitos lógicos incluso si no se requiriese para las herramientas CAD. Dicho simplemente, es interesante y un reto intelectual. En el mundo moderno, desbordante de maquinaria automática ultramoderna, resulta tentador apoyarse en las herramientas como un sustituto de las ideas. Sin embargo, en el diseño de circuitos lógicos, como en cualquier tipo de proceso de diseño, las herramientas basadas en computadoras no suplen la intuición ni la innovación. Las herramientas basadas en computadoras pueden producir buenos diseños de hardware digital sólo cuando las emplea un diseñador que comprende a cabalidad la naturaleza de los circuitos lógicos.

BIBLIOGRAFÍA

1. Semiconductor Industry Association, "National Technology Roadmap for Semiconductors", http://www.semichips.org/
2. Altera Corporation, "Stratix II Field Programmable Gate Arrays", http://www.altera.com
3. Xilinx Corporation, "Virtex-II Field Programmable Gate Arrays", http://www.xilinx.com

4. S. Brown, N. Manjikian, Z. Vranesic, S. Caranci, A. Grbic, R. Grindley, M. Gusat, K. Loveless, Z. Zilic y S. Srbljic, "Experience in Designing a Large-Scale Multiprocessor Using Field-Programmable Devices and Advanced CAD Tools", 33rd IEEE Design Automation Conference, Las Vegas, junio de 1996.

5. A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic y Z. Zilic, "The Design and Implementation of the NUMAchine Multiprocessor", IEEE Design Automation Conference, San Francisco, junio de 1998.

capítulo

2

Introducción a los circuitos lógicos

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- Las funciones y los circuitos lógicos
- El álgebra booleana para manejar las funciones lógicas
- Las compuertas lógicas y la síntesis de circuitos simples
- Las herramientas CAD y el lenguaje VHDL de descripción de hardware

Los circuitos lógicos se estudian principalmente porque se usan en las computadoras digitales. Sin embargo, también constituyen la base de muchos otros sistemas digitales en los que la realización de operaciones aritméticas con números no reviste especial interés. Por ejemplo, en múltiples aplicaciones de control las acciones están determinadas por algunas operaciones lógicas sencillas sobre la información que entra, sin necesidad de hacer muchos cálculos numéricos.

Los circuitos lógicos realizan operaciones con señales digitales y casi siempre se implementan como circuitos electrónicos donde los valores de la señal se restringen a algunos valores discretos. En los circuitos lógicos *binarios* sólo hay dos valores, 0 y 1. En los circuitos lógicos *decimales* hay 10 valores, de 0 a 9. Puesto que el valor de cada señal se representa naturalmente con un dígito, estos circuitos lógicos reciben el nombre de *circuitos digitales*. En contraste, existen los *circuitos analógicos* en los que las señales pueden adquirir una gama discontinua de valores entre un nivel mínimo y uno máximo.

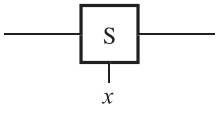
En esta obra se estudian los circuitos binarios, los cuales dominan en la tecnología digital. Esperamos brindar al lector una exposición comprensible de su funcionamiento, de cómo se representan en notación matemática y cómo se diseñan mediante modernas técnicas de diseño automatizado. Empezaremos con la definición de ciertos conceptos básicos relativos a los circuitos lógicos binarios.

2.1 VARIABLES Y FUNCIONES

Los circuitos binarios predominan en los sistemas digitales gracias a su simplicidad, que resulta de restringir las señales para que adopten sólo dos valores posibles. El elemento binario más sencillo es un interruptor de dos estados. Si una variable de entrada x controla un interruptor, entonces se dice que éste se abre si $x = 0$ y se cierra si $x = 1$, como se ilustra en la figura 2.1*a*. Usaremos el símbolo gráfico de la figura 2.1*b* para representar este tipo de interruptores en los diagramas que siguen. Nótese que la entrada de control x se muestra explícitamente en el símbolo. En el capítulo 3 se explica cómo implementar estos interruptores con transistores.

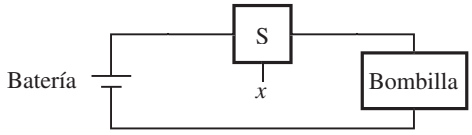
Considérese una aplicación simple de un interruptor, donde éste enciende o apaga una pequeña bombilla. Esta acción se logra con el circuito de la figura 2.2*a*. Una batería proporciona la fuente de poder. La bombilla brilla cuando pasa la corriente necesaria por su filamento, que es una resistencia eléctrica. La corriente fluye cuando el interruptor se cierra; es decir, cuando

a) Dos estados de un interruptor

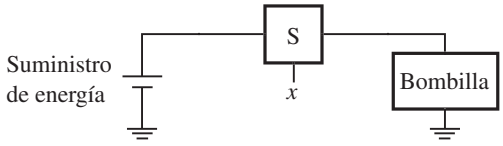


b) Símbolo de un interruptor

Figura 2.1 Interruptor binario.



a) Conexión simple a una batería



b) Uso de una conexión a tierra como trayectoria de regreso

Figura 2.2 Bombilla controlada mediante un interruptor.

$x = 1$. En este ejemplo la entrada que ocasiona el cambio en el comportamiento del circuito es el control x del interruptor. La salida se define como el estado (o condición) de la luz, que se denotará con la letra L. Si la luz se enciende, diremos que $L = 1$; si se apaga, que $L = 0$. Con esta convención es posible describir el estado de la luz como función de la variable de entrada x. Puesto que $L = 1$ si $x = 1$ y $L = 0$ si $x = 0$ puede decirse que

$$L(x) = x$$

Esta sencilla *expresión lógica* describe la salida como función de la entrada. Se dice que $L(x) = x$ es una *función lógica* y que x es una *variable de entrada*.

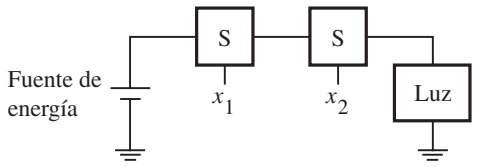
El circuito de la figura 2.2*a* se halla en una linterna ordinaria, donde el interruptor es un dispositivo mecánico sencillo. En un circuito electrónico el interruptor se implementa como un transistor y la luz puede ser un diodo emisor de luz (LED, *light-emitting diode*). Un circuito electrónico recibe la energía de una fuente de cierto voltaje, tal vez 5 voltios. Un lado de la fuente se conecta a tierra, como muestra la figura 2.2*b*. La conexión a tierra también puede usarse como la trayectoria de regreso para la corriente, a fin de cerrar el circuito, lo que se logra conectando un lado de la luz a tierra, como se indica en la figura. Desde luego la luz también puede conectarse con un cable directamente al lado aterrizado de la fuente de poder, como se advierte en la figura 2.2*a*.

Considérese ahora la posibilidad de usar dos interruptores para controlar el estado de la luz. Sean x_1 y x_2 sus entradas de control. Los interruptores pueden conectarse en serie o en paralelo, como se muestra en la figura 2.3. Si se usa conexión en serie la luz se encenderá sólo si ambos interruptores están cerrados. Si uno está abierto, la luz estará apagada. Este comportamiento puede describirse con la expresión

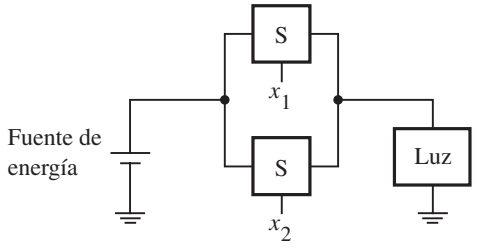
$$L(x_1, x_2) = x_1 \cdot x_2$$

donde $L = 1$ si $x_1 = 1$ y $x_2 = 1$,

$L = 0$ de otro modo.



a) La función lógica AND (conexión en serie)



b) La función lógica OR (conexión en paralelo)

Figura 2.3 Dos funciones básicas.

El símbolo "·" es el *operador AND*, y se dice que el circuito de la figura 2.3*a* implementa la *función lógica AND*.

En la figura 2.3*b* se presenta la conexión en paralelo de dos interruptores. En este caso la luz se encenderá si cualquiera de los interruptores, x_1 o x_2, se cierra, o si ambos se cierran. La luz se apagará sólo si los dos interruptores están abiertos. Este comportamiento puede expresarse como

$$\begin{aligned} & L(x_1, x_2) = x_1 + x_2 \\ \text{donde} \quad & L = 1 \text{ si } x_1 = 1 \text{ o } x_2 = 1 \text{ o si } x_1 = x_2 = 1, \\ & L = 0 \text{ si } x_1 = x_2 = 0. \end{aligned}$$

El símbolo + es el *operador OR* y se dice que el circuito de la figura 2.3*b* implementa la *función lógica OR*.

En las expresiones anteriores para AND y OR, la salida $L(x_1, x_2)$ es una función lógica con variables de entrada x_1 y x_2. Las funciones AND y OR son dos de las funciones lógicas más importantes. Junto con algunas otras funciones simples se usan como los bloques fundamentales de la implementación de todos los circuitos lógicos. En la figura 2.4 se muestra cómo usar tres interruptores para controlar la luz de forma más compleja. Esta conexión serie-paralelo de interruptores realiza la función lógica

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

La luz se enciende si $x_3 = 1$ y, al mismo tiempo, al menos una de las entradas x_1 o x_2 es igual a 1.

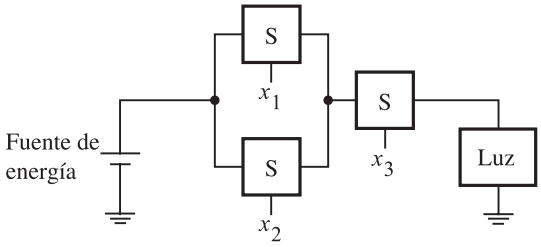


Figura 2.4 Conexión serie-paralelo.

2.2 INVERSIÓN

Hasta el momento hemos supuesto que cierta acción positiva, como encender la luz, tiene lugar cuando se cierra un interruptor. Es igualmente interesante y útil considerar la posibilidad de que suceda una acción positiva cuando se abre un interruptor. Supóngase que conectamos la luz como se muestra en la figura 2.5. En este caso el interruptor se conecta en paralelo con la luz, en lugar de en serie. En consecuencia un interruptor cerrado ocasionará un cortocircuito y evitará que la corriente pase por él. Nótese que hemos incluido en este circuito un resistor adicional para garantizar que el interruptor cerrado no causará un cortocircuito en la fuente de energía. La luz se encenderá cuando el interruptor se abra. Formalmente, este comportamiento funcional se expresa como

$$L(x) = \overline{x}$$

$$\text{donde} \quad L = 1 \text{ si } x = 0,$$

$$L = 0 \text{ si } x = 1.$$

El valor de esta función es el inverso del valor de la variable de entrada. En lugar de utilizar la palabra *inverso*, es más común usar el término *complemento*. Por tanto se dice que $L(x)$ es un complemento de x en este ejemplo. Otro término empleado con frecuencia para la misma operación es *operación NOT*. Diversas notaciones se usan para indicar el complemento. En la expresión precedente se coloca una barra sobre la x. Quizá esta notación sea la mejor desde un ángulo visual. Sin embargo, cuando se requieren complementos en las expresiones que se escriben con

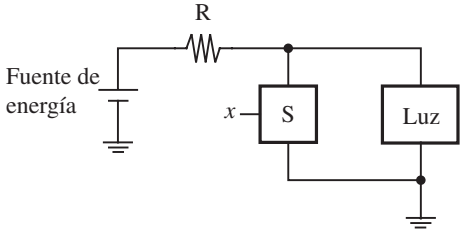


Figura 2.5 Un circuito inversor.

el teclado de una computadora, lo que a menudo sucede cuando se emplean herramientas CAD, resulta impráctico usar barras superiores. En su lugar se coloca un apóstrofe después de la variable, o el signo de exclamación (!), o el tilde (~), o la palabra NOT frente a la variable para denotar la complementación. Por ende, las expresiones que siguen son equivalentes:

$$\overline{x} = x' = !x = \sim x = \text{NOT } x$$

La operación complemento puede aplicarse a una sola variable o a operaciones más complejas. Por ejemplo, si

$$f(x_1, x_2) = x_1 + x_2$$

entonces el complemento de f es

$$\overline{f}(x_1, x_2) = \overline{x_1 + x_2}$$

Esta expresión produce el valor lógico 1 sólo cuando ni x_1 ni x_2 son iguales a 1; es decir: cuando $x_1 = x_2 = 0$. De nuevo las notaciones que siguen son equivalentes:

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2) = \text{NOT } (x_1 + x_2)$$

2.3 TABLAS DE VERDAD

Hemos presentado las tres operaciones lógicas más básicas —AND, OR y complemento— relacionándolas con circuitos sencillos construidos con interruptores. Este enfoque confiere a tales operaciones cierto "significado físico". Las mismas operaciones también pueden definirse en forma de tabla, llamada *tabla de verdad*, como se muestra en la figura 2.6. Las primeras dos columnas (a la izquierda de la línea vertical doble) proporcionan las cuatro posibles combinaciones de valores lógicos que las variables x_1 y x_2 pueden tener. La siguiente columna define la operación AND para cada combinación de valores de x_1 y x_2, y la última columna define la operación OR. Puesto que con frecuencia es necesario hacer referencia a "combinaciones de valores lógicos" aplicados a algunas variables, se adoptará un término más corto, *valoración*, para denotar tal combinación de valores lógicos.

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figura 2.6 Tabla de verdad para las operaciones AND y OR.

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figura 2.7 Operaciones AND y OR para tres entradas.

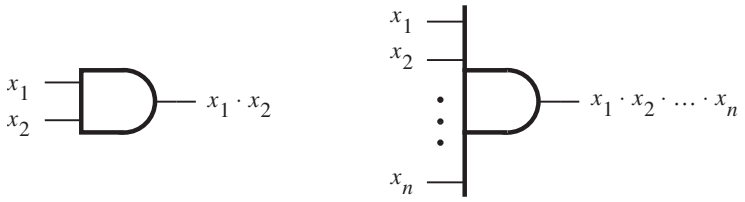
Las tablas de verdad son un auxiliar útil para describir información relacionada con funciones lógicas. En este libro se utilizan para definir funciones específicas y demostrar la validez de ciertas relaciones funcionales. Las tablas de verdad pequeñas son fáciles de manejar. Sin embargo, crecen exponencialmente en tamaño con el número de variables. Una tabla de verdad de tres variables de entrada tiene ocho filas porque hay ocho posibles valoraciones para esas variables. La figura 2.7 proporciona una tabla semejante, que define las funciones AND y OR para tres entradas. Para cuatro variables de entrada, la tabla de verdad tiene 16 filas, etc. En general, para n variables de entrada, la tabla de verdad tiene 2^n filas.

Las operaciones AND y OR pueden extenderse a n variables. Una función AND de variables $x_1, x_2, \ldots, x_n$ tiene el valor 1 sólo si todas las n variables son iguales a 1. Una función OR de variables $x_1, x_2, \ldots, x_n$ tiene el valor 1 si una o más de las variables es igual a 1.

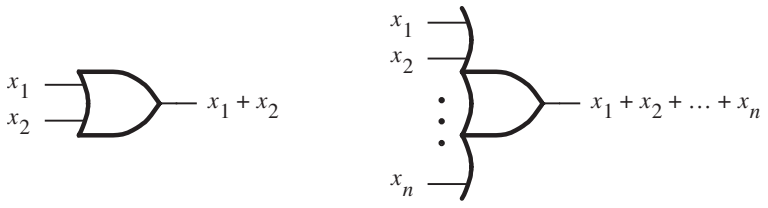
2.4 COMPUERTAS LÓGICAS Y CIRCUITOS

Las tres operaciones lógicas básicas expuestas en las secciones previas pueden usarse para implementar funciones lógicas de cualquier complejidad. La puesta en marcha de una función compleja puede requerir muchas de estas operaciones básicas. Cada operación lógica puede implementarse electrónicamente con transistores, lo que resulta en un elemento de circuito denominado *compuerta lógica*. Una compuerta lógica tiene una o más entradas y una salida que es función de éstas. Suele ser conveniente describir un circuito lógico trazando un diagrama del circuito, o *esquema*, que consta de símbolos gráficos que representan las compuertas lógicas. Los símbolos gráficos de las compuertas AND, OR y NOT se muestran en la figura 2.8. En el lado izquierdo se indica cómo dibujar las compuertas AND y OR cuando hay pocas entradas. En el lado derecho se muestra cómo aumentan los símbolos para dar cabida a un mayor número de entradas. En el capítulo 3 se explica la manera de construir las compuertas lógicas con transistores.

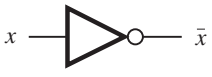
Un circuito más grande se implementa mediante una *red* de compuertas. Por ejemplo, la función lógica de la figura 2.4 puede realizarse mediante la red de la figura 2.9. La complejidad de una red tiene un efecto directo en su costo. Como siempre es deseable reducir el costo de los



a) Compuertas AND



b) Compuertas OR



c) Compuertas NOT

Figura 2.8 Las compuertas básicas.

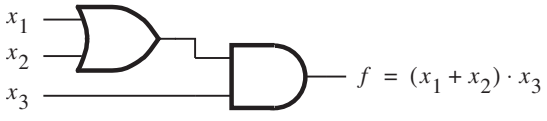


Figura 2.9 La función de la figura 2.4.

productos fabricados, es importante encontrar formas de implementar los circuitos lógicos lo más barato posible. Páginas adelante se verá que una función lógica puede implementarse con redes diferentes, algunas de las cuales son más simples que otras; por tanto, resulta prudente buscar soluciones que representen el costo mínimo.

En el lenguaje técnico, una red de compuertas recibe el nombre de *red lógica* o, simplemente, *circuito lógico*. Utilizaremos estos términos como sinónimos.

2.4.1 ANÁLISIS DE UNA RED LÓGICA

Un diseñador de sistemas digitales enfrenta dos conflictos básicos. Debe ser posible determinar la función que realiza una red lógica existente. Esta tarea se conoce como proceso de *análisis*. La tarea inversa de diseñar una nueva red que desempeñe cierto comportamiento funcional se denomina proceso de *síntesis*. El proceso de análisis es más bien directo y mucho más sencillo que el de síntesis.

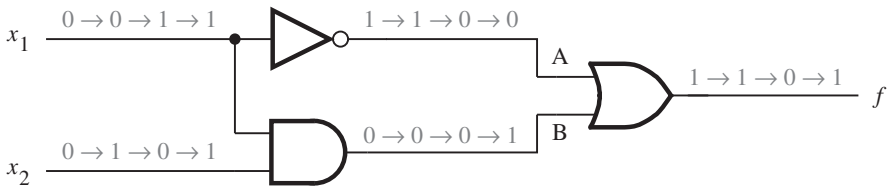
En la figura 2.10*a* se muestra una red simple formada por tres compuertas. A fin de determinar su comportamiento funcional considérese lo que ocurre si se aplican todas las señales de entrada posibles. Supóngase que se inicia $x_1 = x_2 = 0$. Esto obliga a la salida de la compuerta NOT a ser igual a 1 y a la salida de la compuerta AND a ser 0. Puesto que una de las entradas a la compuerta OR es 1, la salida de esta compuerta será 1. Por tanto, $f = 1$ si $x_1 = x_2 = 0$. Si se hace $x_1 = 0$ y $x_2 = 1$, entonces no ocurrirá cambio en el valor de *f*, pues las salidas de las compuertas NOT y AND seguirán siendo 1 y 0 respectivamente. A continuación, si se aplica $x_1 = 1$ y $x_2 = 0$, entonces la salida de la compuerta NOT cambiará a 0, mientras que la de la compuerta AND continuará siendo 0. Ambas entradas a la compuerta OR serán iguales a 0; por ende, el valor de *f* será 0. Finalmente, sea $x_1 = x_2 = 1$. Entonces la salida de la compuerta AND será 1, lo que produce que *f* sea igual a 1. La explicación verbal puede resumirse en la forma de la tabla de verdad de la figura 2.10*b*.

Diagramas de tiempo

El comportamiento de la red de la figura 2.10*a* se determinó al considerar los cuatro posibles valores de las entradas x_1 y x_2. Supóngase que las señales correspondientes a esas valoraciones se aplican a la red en el orden que acabamos de describir; esto es: $(x_1, x_2) = (0, 0)$, seguido de (0, 1), (1, 0) y (1, 1). Luego, los cambios en las señales en varios puntos de la red serían como se indica en gris en la figura. La misma información puede presentarse en forma gráfica, conocida como *diagrama de tiempo*, como se muestra en la figura 2.10*c*. El tiempo corre de izquierda a derecha y la valoración de cada entrada se mantiene cierto periodo fijo. En la figura se muestran las formas de onda de las entradas y salidas de la red, así como de las señales internas en los puntos *A* y *B*.

El diagrama de tiempo de la figura 2.10*c* indica que los cambios en las formas de onda de los puntos *A* y *B*, y la salida *f* tienen lugar instantáneamente cuando los valores de las entradas x_1 y x_2 cambian. Estas formas de onda idealizadas se basan en la suposición de que las compuertas lógicas responden a cambios en sus entradas en tiempo cero. Tales diagramas de tiempo son útiles para indicar el *comportamiento funcional* de los circuitos lógicos. Sin embargo, las compuertas lógicas prácticas se implementan con circuitos electrónicos que requieren cierto tiempo para cambiar sus estados. Por tanto, hay un retardo entre un cambio en los valores de entrada y el cambio correspondiente en el valor de salida de una compuerta. En capítulos próximos emplearemos diagramas de tiempo que incorporan tales retardos.

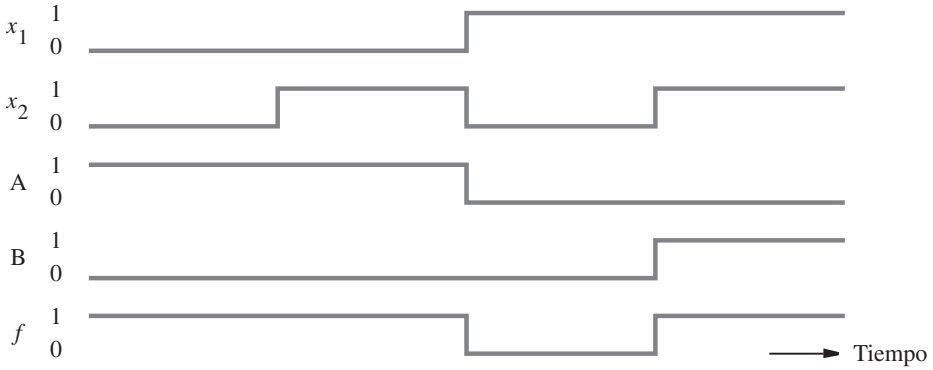
Los diagramas de tiempo sirven para muchos fines. Describen el comportamiento de un circuito lógico en una forma que es posible observar cuando se pone a prueba el circuito con instrumentos como analizadores lógicos y osciloscopios. Además a menudo se generan mediante herramientas CAD con objeto de mostrar al diseñador cómo se comporta un circuito antes de implementarlo electrónicamente en la realidad. Más adelante estudiaremos las herramientas CAD que se usarán a lo largo del libro.



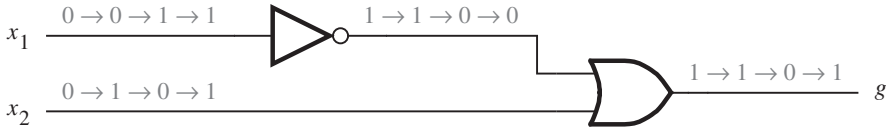
a) Red que implementa $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

b) Tabla de verdad



c) Diagrama de tiempo



d) Red que implementa $g = \bar{x}_1 + x_2$

Figura 2.10 Ejemplo de redes lógicas.

Redes funcionalmente equivalentes

Considérese ahora la red de la figura 2.10*d*. Al realizar el mismo análisis se determina que la salida g cambia exactamente de la misma manera en que lo hace f en la parte *a*) de la figura. Por consiguiente, $g(x_1, x_2) = f(x_1, x_2)$, que indica que los dos circuitos son iguales en términos funcionales; la tabla de verdad de la figura 2.10*b* representa el comportamiento de salida de

ambos circuitos. Como los dos cumplen la misma función, es lógico utilizar la más simple, cuya implementación es menos costosa.

En general, una función lógica puede implementarse con diferentes circuitos, que quizá tengan distintos costos. Esto da lugar a una pregunta importante: ¿cómo se determina cuál es la mejor forma de implementar cierta función? Hay numerosas técnicas para sintetizar funciones lógicas. Estudiaremos los principales enfoques para ello en el capítulo 4. Por ahora cabe señalar que se precisa cierta manipulación para transformar el circuito más complejo de la figura 2.10*a* en el de la figura 2.10*d*. Puesto que $f(x_1, x_2) = \overline{x}_1 + x_1 \cdot x_2$ y $g(x_1, x_2) = \overline{x}_1 + x_2$, debe haber ciertas reglas que puedan aplicarse para demostrar la equivalencia

$$\overline{x}_1 + x_1 \cdot x_2 = \overline{x}_1 + x_2$$

Ya establecimos esta equivalencia con el análisis detallado de los dos circuitos y la elaboración de la tabla de verdad. No obstante, puede obtenerse el mismo resultado mediante manipulación algebraica de expresiones lógicas. En la siguiente sección se explicará un método matemático para tratar las funciones lógicas, el cual brinda las bases de las técnicas de diseño modernas.

2.5 ÁLGEBRA BOOLEANA

En 1849, George Boole publicó un esquema de la descripción algebraica de los procesos relativos al pensamiento y el razonamiento lógicos [1]. Luego ese esquema y sus posteriores refinamientos recibieron el nombre de *álgebra booleana*. Fue casi 100 años después que esta álgebra halló aplicación en la ingeniería. A fines de la década de 1930, Claude Shannon demostró que el álgebra booleana constituye un medio eficaz para describir circuitos construidos con interruptores [2]; por tanto, esta algebra sirve para describir circuitos lógicos. En este apartado veremos que el álgebra booleana constituye una poderosa herramienta para diseñar y analizar circuitos lógicos. El lector advertirá que sienta las bases de gran parte de la tecnología digital de nuestros días.

Axiomas del álgebra booleana

Como cualquier álgebra, la booleana se basa en un conjunto de reglas derivadas a partir de un pequeño número de suposiciones fundamentales que reciben el nombre de *axiomas*. Supóngase que el álgebra booleana *B* comprende elementos que toman uno de dos valores, 0 y 1. Supóngase asimismo que los axiomas siguientes son verdaderos:

1*a*. $0 \cdot 0 = 0$
1*b*. $1 + 1 = 1$
2*a*. $1 \cdot 1 = 1$
2*b*. $0 + 0 = 0$
3*a*. $0 \cdot 1 = 1 \cdot 0 = 0$
3*b*. $1 + 0 = 0 + 1 = 1$
4*a*. Si $x = 0$, entonces $\overline{x} = 1$
4*b*. Si $x = 0$, entonces $\overline{x} = 0$

Teoremas de una sola variable

A partir de los axiomas pueden definirse ciertas reglas para usar las variables individuales. A menudo esas reglas se denominan *teoremas*. Si x es una variable en B, entonces se cumplen los teoremas siguientes:

5*a*. $x \cdot 0 = 0$
5*b*. $x + 1 = 1$
6*a*. $x \cdot 1 = x$
6*b*. $x + 0 = x$
7*a*. $x \cdot x = x$
7*b*. $x + x = x$
8*a*. $x \cdot \overline{x} = 0$
8*b*. $x + \overline{x} = 1$
9. $\overline{\overline{x}} = x$

Es fácil probar la validez de estos teoremas mediante inducción perfecta; es decir, mediante la sustitución de los valores $x = 0$ y $x = 1$ en las expresiones y la aplicación de los axiomas anteriores. Por ejemplo, en el teorema 5*a*, si $x = 0$ entonces el teorema afirma que $0 \cdot 0 = 0$, lo que es cierto de acuerdo con el axioma 1*a*. De manera similar, si $x = 1$ entonces el teorema 5*a* afirma que $1 \cdot 0 = 0$, que también es cierto según el axioma 3*a*. El lector debe verificar que los teoremas 5*a* a 9 pueden comprobarse de este modo.

Dualidad

Nótese que hemos numerado por pares los axiomas y los teoremas de una sola variable. Lo hicimos para reflejar la importancia del *principio de dualidad*. Dada una expresión lógica, su *dual* se obtiene sustituyendo todos los operadores $+$ con operadores $\cdot$, y viceversa, y sustituyendo todos los 0 con 1, y viceversa. El dual de cualquier proposición verdadera (axioma o teorema) en álgebra booleana también es una proposición verdadera. Si bien en este punto de la explicación el lector no advertirá por qué la dualidad es un concepto útil, le quedará claro más adelante, cuando se muestre que la dualidad implica la existencia de al menos dos formas de expresar toda función lógica con álgebra booleana. Con frecuencia, una expresión conduce a una implementación física más simple que la otra y por tanto es preferible.

Propiedades de dos y tres variables

Para que sea posible tratar con varias variables es útil definir algunas identidades algebraicas de dos y tres variables. Para cada una de ellas también se proporciona su versión dual. Estas identidades suelen denominarse *propiedades*. Se conocen por los nombres que se indican a continuación. Si x, y y z son las variables en B, entonces se cumplen las siguientes propiedades:

10*a*.	$x \cdot y = y \cdot x$	*Conmutativa*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Asociativa*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributiva*
12*b*.	$x + y \cdot z = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absorción*

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LI		LD		

Figura 2.11 Prueba del teorema de DeMorgan en 15*a*.

13*b*. $x \cdot (x + y) = x$

14*a*. $x \cdot y + x \cdot \overline{y} = x$ *Combinación*

14*b*. $(x + y) \cdot (x + \overline{y}) = x$

15*a*. $\overline{x \cdot y} = \overline{x} + \overline{y}$ *Teorema de DeMorgan*

15*b*. $\overline{x + y} = \overline{x} \cdot \overline{y}$

16*a*. $x + \overline{x} \cdot y = x + y$

16*b*. $x \cdot (\overline{x} + y) = x \cdot y$

17*a*. $x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$ *Consenso*

17*b*. $(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$

De nuevo, es posible probar la validez de estas propiedades mediante inducción perfecta o por manipulación algebraica. En la figura 2.11 se indica cómo usar la inducción perfecta para demostrar el teorema de DeMorgan por medio de una tabla de verdad. La evaluación de los lados izquierdo y derecho de la identidad en 15*a* da el mismo resultado.

Hemos enumerado varios axiomas, teoremas y propiedades. No todos ellos son necesarios para definir el álgebra booleana. Por ejemplo, si suponemos que las operaciones + y · están definidas basta incluir los teoremas 5 y 8 y las propiedades 10 y 12. Éstos a veces se refieren como *postulados básicos de Huntington* [3]. El resto de las identidades puede derivarse de ellos.

Los axiomas, teoremas y propiedades anteriores proveen la información necesaria para realizar la manipulación algebraica de expresiones más complejas.

Ejemplo 2.1

Demostremos la validez de la ecuación lógica

$$(x_1 + x_3) \cdot (\overline{x}_1 + \overline{x}_3) = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

El miembro izquierdo de la ecuación se manipula como sigue. Al aplicar la propiedad distributiva, 12*a*, se obtiene

$$\text{LI} = (x_1 + x_3) \cdot \overline{x}_1 + (x_1 + x_3) \cdot \overline{x}_3$$

Al aplicar de nuevo la propiedad distributiva se obtiene

$$\text{LI} = x_1 \cdot \overline{x}_1 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + x_3 \cdot \overline{x}_3$$

Nótese que la propiedad distributiva permite aplicar la operación AND en los términos entre paréntesis en una forma análoga a la multiplicación del álgebra ordinaria. Ahora, de acuerdo con el teorema 8*a*, los términos $x_1 \cdot \overline{x}_1$ y $x_3 \cdot \overline{x}_3$ son ambos iguales a 0. Por tanto,

$$\text{LI} = 0 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + 0$$

Con base en 6*b* se sigue que

$$\text{LI} = x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3$$

Finalmente, al usar la propiedad conmutativa, 10*a* y 10*b*, esto se convierte en

$$\text{LI} = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

que es lo mismo que el miembro derecho de la ecuación inicial.

Ejemplo 2.2 Considérese la ecuación lógica

$$x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 = \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2$$

El miembro izquierdo puede manipularse del modo siguiente

$$\begin{aligned}
\text{LI} &= x_1 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_2 \cdot x_3 && \text{al aplicar } 10b \\
&= x_1 \cdot (\overline{x}_3 + x_3) + \overline{x}_2 \cdot (\overline{x}_3 + x_3) && \text{al aplicar } 12a \\
&= x_1 \cdot 1 + \overline{x}_2 \cdot 1 && \text{al aplicar } 8b \\
&= x_1 + \overline{x}_2 && \text{al aplicar } 6a
\end{aligned}$$

El miembro derecho se manipula como

$$\begin{aligned}
\text{LD} &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot (x_2 + \overline{x}_2) && \text{al aplicar } 12a \\
&= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot 1 && \text{al aplicar } 8b \\
&= \overline{x}_1 \cdot \overline{x}_2 + x_1 && \text{al aplicar } 6a \\
&= x_1 + \overline{x}_1 \cdot \overline{x}_2 && \text{al aplicar } 10b \\
&= x_1 + \overline{x}_2 && \text{al aplicar } 16a
\end{aligned}$$

Al ser posible manipular ambos miembros de la ecuación inicial para llegar a expresiones idénticas se establece la validez de la ecuación. Nótese que la misma función lógica se representa mediante el miembro izquierdo o el derecho de la ecuación anterior:

$$\begin{aligned}
f(x_1, x_2, x_3) &= x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 \\
&= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2
\end{aligned}$$

Como resultado de la manipulación se halló una expresión mucho más simple

$$f(x_1, x_2, x_3) = x_1 + \overline{x}_2$$

que representa la misma función. Esta expresión más simple resultaría en un circuito lógico de menor costo que podría usarse para implementar la función.

Los ejemplos 2.1 y 2.2 ilustran el propósito de los axiomas, teoremas y propiedades como un mecanismo de manipulación algebraica. Incluso estos ejemplos simples sugieren que no es práctico tratar de esta forma con expresiones sumamente complejas. Sin embargo, dichos teoremas y propiedades ofrecen la base para automatizar la síntesis de las funciones lógicas en las herramientas CAD. Para comprender qué puede lograrse con tales herramientas el diseñador ha de estar consciente de los conceptos fundamentales.

2.5.1 LOS DIAGRAMAS DE VENN

Antes se sugirió que la inducción perfecta puede usarse para comprobar los teoremas y las propiedades. Este procedimiento es bastante tedioso y no muy informativo desde el punto de vista conceptual. Hay un auxiliar visual sencillo que sirve para este propósito. Se llama *diagrama de Venn* y es probable que el lector encuentre que le ofrece una explicación más intuitiva de cómo dos expresiones pueden ser equivalentes.

Tradicionalmente los diagramas de Venn se usan en matemáticas para ilustrar de modo gráfico varias operaciones y relaciones en el álgebra de conjuntos. Un conjunto s es una colección de elementos que se dice son miembros de s. En el diagrama de Venn los elementos de un conjunto se representan mediante el contorno cerrado de una figura geométrica, digamos un cuadrado, un círculo o una elipse. Por ejemplo, en un universo N de enteros de 1 a 10 el conjunto de números pares es $E = \{2, 4, 6, 8, 10\}$. Un contorno que representa a E encierra los números pares. Los nones forman el complemento de E; por tanto, el área fuera del contorno representa $E = \{1, 3, 5, 7, 9\}$.

Como en el álgebra booleana sólo hay dos valores (elementos) en el universo, $B = \{0, 1\}$, se dice que el área dentro de un contorno que corresponde a un conjunto s denota que $s = 1$, mientras que el área fuera del contorno denota que $s = 0$. En el diagrama sombrearemos el área donde $s = 1$. En la figura 2.12 se muestra el concepto del diagrama de Venn. Un cuadrado representa el universo B. En los incisos *a*) y *b*) de la figura se advierte la representación de las constantes 1 y 0. Un círculo representa una variable, digamos x, de modo que el área del círculo corresponde a $x = 1$, mientras que el área fuera del círculo corresponde a $x = 0$. Esto se ilustra en el inciso *c*). Una expresión que comprende una o más variables se describe mediante el sombreado del área donde el valor de la expresión es igual a 1. En el inciso *d*) se indica cómo representar el complemento de x.

Para representar dos variables, x y y, se trazan dos círculos que se traslapan. El área de traslape representa el caso donde $x = y = 1$, es decir, el AND de x y y, como se muestra en el inciso *e*). Puesto que esta área común se forma por las partes de x y y que se intersecan, la operación AND recibe formalmente el nombre de *intersección* de x y y. En el inciso *f*) se ilustra la operación OR, donde $x + y$ representa el área total dentro de ambos círculos, o sea, donde al menos x o y es igual a 1. Ya que con esto se combinan las áreas de los círculos, a menudo la operación OR formalmente se llama unión de x y y.

En el inciso *g*) se describe el término producto $x \cdot \overline{y}$, representado por la intersección del área de x con la de $\overline{y}$. El inciso *h*) presenta un ejemplo de tres variables; la expresión $x \cdot y + z$ es la unión del área de z con la de la intersección de x y y.

Para ver cómo se utilizan los diagramas de Venn a fin de comprobar la equivalencia de dos expresiones demostraremos la validez de la propiedad distributiva, 12*a*, de la sección 2.5. En la figura 2.13 aparece la construcción de los miembros izquierdo y derecho de la identidad que define la propiedad

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

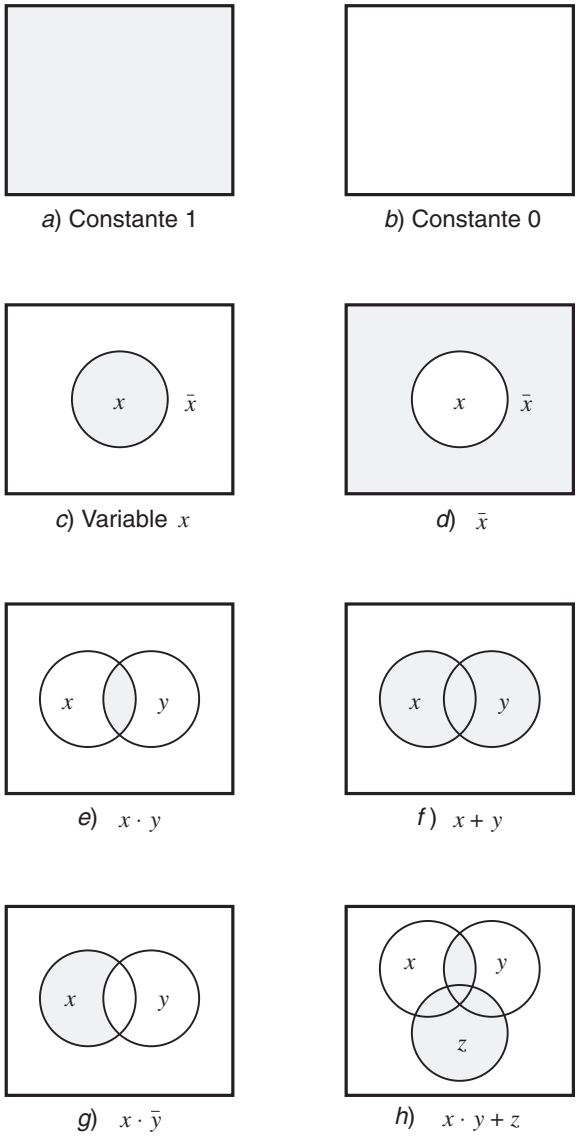


Figura 2.12 Representación de diagramas de Venn.

El inciso *a*) muestra el área donde $x = 1$. El *b*) indica el área de $y + z$. En el *c*) se proporciona el diagrama para $x \cdot (y + z)$, la intersección de las áreas sombreadas en los incisos *a*) y *b*). El miembro derecho se construye en los incisos *d*), *e*) y *f*). Los incisos *d*) y *e*) describen los términos $x \cdot y$ y $x \cdot z$, respectivamente. La unión de las áreas sombreadas en estos dos diagramas corresponde entonces a la expresión $x \cdot y + x \cdot z$, como se observa en el inciso *f*). Como las áreas sombreadas en los incisos *c*) y *f*) son idénticas, se deduce que la propiedad distributiva es válida.

Como otro ejemplo, considérese la identidad

$$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$$

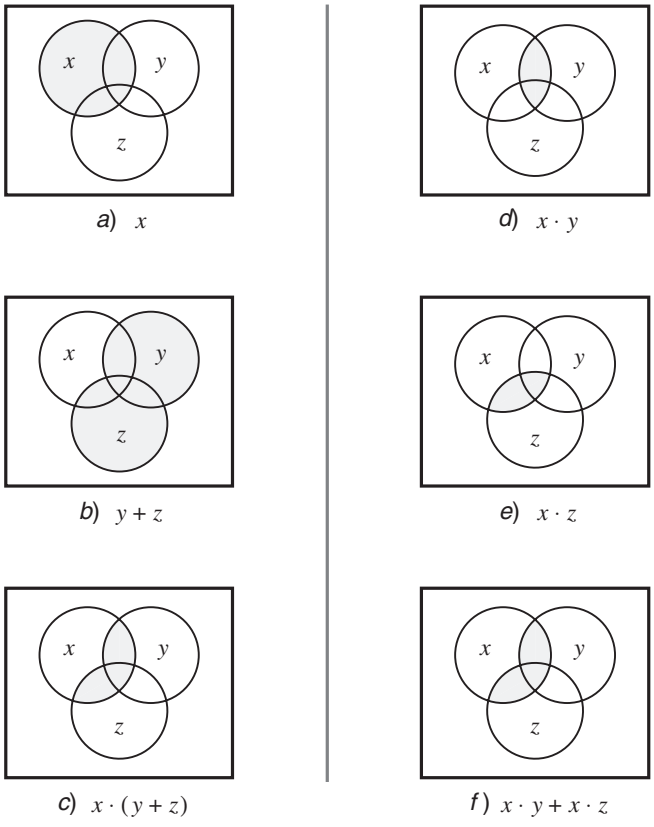


Figura 2.13 Comprobación de la propiedad distributiva $x \cdot (y + z) = x \cdot y + x \cdot z$.

que se ilustra en la figura 2.14. Nótese que esta identidad establece que el término $y \cdot z$ está completamente cubierto por los términos $x \cdot y$ y $\overline{x} \cdot z$; en consecuencia, este término puede omitirse.

El lector debe usar los diagramas de Venn para probar algunas otras identidades. Es en particular instructivo probar la validez del teorema de DeMorgan de esta manera.

2.5.2 NOTACIÓN Y TERMINOLOGÍA

El álgebra booleana se basa en las operaciones AND y OR. En el texto hemos adoptado los símbolos · y + para denotarlas; se trata de los símbolos de las conocidas operaciones aritméticas de multiplicación y suma. Entre las operaciones booleanas y las aritméticas hay una similitud considerable, principal razón por la que se usan los mismos símbolos. De hecho, cuando únicamente hay dígitos solos todo se reduce a una diferencia significativa; en la aritmética ordinaria el resultado de 1 + 1 es igual a 2, mientras que el álgebra booleana es igual a 1, como lo define el teorema 7*b* de la sección 2.5.

Cuando se trabaja con circuitos digitales el símbolo + casi siempre representa la operación OR. No obstante, cuando la tarea supone el diseño de circuitos lógicos que realizan operaciones

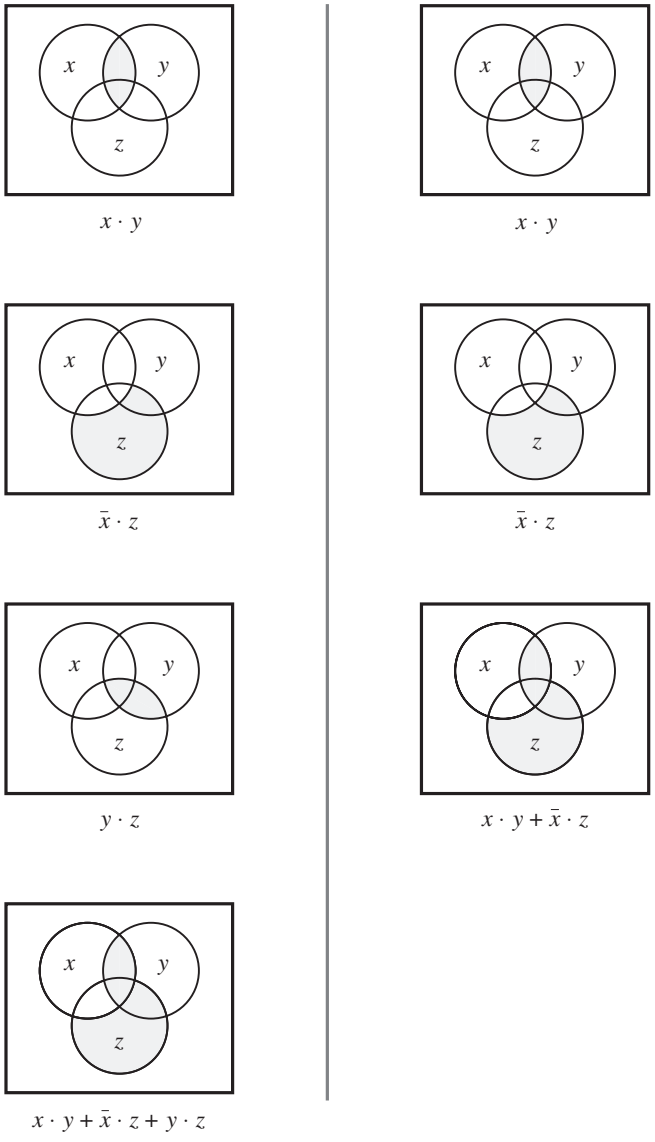


Figura 2.14 Comprobación de $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$.

aritméticas es posible que se cree cierta confusión en torno a su uso. Para evitarla existe un conjunto de símbolos diferentes para las operaciones AND y OR. Es muy común usar el símbolo $\wedge$ para denotar la operación AND y $\vee$ para la operación OR. Por ende, en lugar de $x_1 \cdot x_2$, puede escribirse $x_1 \wedge x_2$, y en vez de $x_1 + x_2$, se escribe $x_1 \vee x_2$.

Por la similitud con las operaciones de suma y multiplicación aritméticas, las operaciones OR y AND con frecuencia se denominan operaciones de *suma* y *producto lógicos*. Por tanto, $x_1 + x_2$ es la suma lógica de x_1 y x_2, y $x_1 \cdot x_2$ es el producto lógico de x_1 y x_2. En lugar de decir

"producto lógico" y "suma lógica" suele decirse simplemente "producto" y "suma". Por ende, la expresión

$$x_1 \cdot \overline{x}_2 \cdot x_3 + \overline{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \overline{x}_4$$

es una suma de tres productos, en tanto que

$$(\overline{x}_1 + x_3) \cdot (x_1 + \overline{x}_3) \cdot (\overline{x}_2 + x_3 + x_4)$$

es un producto de tres sumas.

2.5.3 PRECEDENCIA DE LAS OPERACIONES

Con las tres operaciones básicas —AND, OR y NOT— es posible construir un número infinito de expresiones lógicas. Se emplean paréntesis para indicar el orden en que las operaciones deben realizarse, pero para no usarlos en exceso la precedencia de las operaciones básicas se define con otra convención. Ésta afirma que, en ausencia de paréntesis, las operaciones de una expresión lógica deben realizarse en el orden NOT, AND y luego OR. Por consiguiente, en la expresión

$$x_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2$$

primero hay que generar los complementos de x_1 y x_2. Después se forman los términos producto $x_1 \cdot x_2$ y $\overline{x}_1 \cdot \overline{x}_2$, seguidos por la suma de los dos términos producto. Obsérvese que sin esta convención habría que utilizar paréntesis para lograr el mismo resultado, como sigue:

$$(x_1 \cdot x_2) + ((\overline{x}_1) \cdot (\overline{x}_2))$$

Finalmente, para simplificar la presentación de las expresiones lógicas se omite el operador $\cdot$ cuando no existe ambigüedad. Por tanto, la expresión anterior puede escribirse como

$$x_1 x_2 + \overline{x}_1 \overline{x}_2$$

Seguiremos este estilo a lo largo del libro.

2.6 LA SÍNTESIS CON COMPUERTAS AND, OR Y NOT

Con estas ideas básicas ahora podemos intentar la implementación de funciones arbitrarias mediante las compuertas AND, OR y NOT. Supóngase que queremos diseñar un circuito lógico con dos entradas, x_1 y x_2. Digamos que x_1 y x_2 representan los estados de dos interruptores, cualquiera de los cuales puede estar abierto (0) o cerrado (1). La función del circuito es revisar continuamente el estado de los interruptores (x_1, x_2) y producir un valor lógico de salida 1 siempre que éstos se hallen en los estados (0, 0), (0, 1) o (1, 1). Si el estado de los interruptores es (1, 0) la salida debe ser 0. Otra manera de describir el comportamiento funcional requerido de este circuito es que la salida debe ser igual a 0 si se cierra el interruptor x_1 y se abre el x_2; de otro modo, la salida debe ser 1. El comportamiento requerido puede expresarse por medio de una tabla de verdad como la que se muestra en la figura 2.15.

Un posible procedimiento para diseñar un circuito lógico que implemente la tabla de verdad es crear un término producto que tenga un valor de 1 por cada valoración para que la función

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

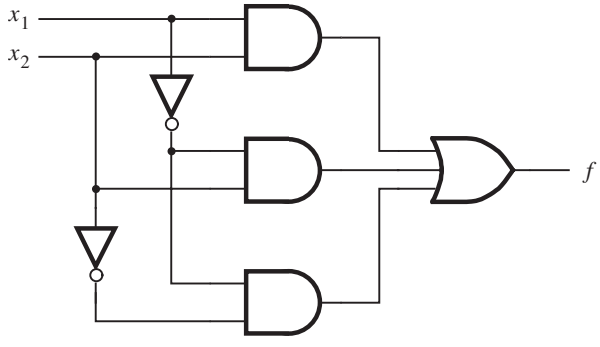
Figura 2.15 Función que va a sintetizarse.

de salida f deba ser 1. Luego podemos tomar una suma lógica de estos términos producto para realizar f. Empezaremos con la cuarta fila de la tabla de verdad, que corresponde a $x_1 = x_2 = 1$. El término producto que es igual a 1 para esta valoración es $x_1 \cdot x_2$, que es justo la función AND de x_1 y x_2. A continuación considérese la primera fila de la tabla, para la que $x_1 = x_2 = 0$. Para esta valoración, el valor 1 se produce por el término producto $\overline{x}_1 \cdot \overline{x}_2$. De manera similar, la segunda fila conduce al término $\overline{x}_1 \cdot x_2$. Por ende, f puede realizarse como

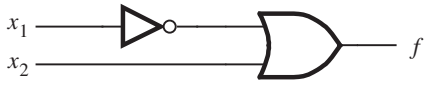
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

El circuito lógico correspondiente a esta expresión se muestra en la figura 2.16*a*.

Aunque este circuito implementa f correctamente, no es el más simple. Para encontrar uno más simple puede manipularse la expresión obtenida mediante los teoremas y propiedades de la sección 2.5. De acuerdo con el teorema 7*b*, es posible duplicar cualquier término de una



a) Suma canónica de productos



b) Realización de costo mínimo

Figura 2.16 Dos implementaciones de la función de la figura 2.15.

expresión de suma lógica. Si duplicamos el tercer término producto la expresión anterior se convierte en

$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + \overline{x}_1x_2$$

Con la propiedad conmutativa 10*b* para intercambiar los términos producto segundo y tercero se obtiene

$$f(x_1, x_2) = x_1x_2 + \overline{x}_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

Ahora la propiedad distributiva 12*a* permite escribir

$$f(x_1, x_2) = (x_1 + \overline{x}_1)x_2 + \overline{x}_1(\overline{x}_2 + x_2)$$

Al aplicar el teorema 8*b* se obtiene

$$f(x_1, x_2) = 1 \cdot x_2 + \overline{x}_1 \cdot 1$$

Por último, el teorema 6*a* conduce a

$$f(x_1, x_2) = x_2 + \overline{x}_1$$

El circuito descrito por esta expresión se presenta en la figura 2.16*b*. Resulta obvio que su costo es mucho menor que el del circuito del inciso *a*) de la figura.

Con este ejemplo sencillo se ilustran dos cosas. Primero, es posible obtener una implementación directa de una función si se usa un término producto (compuerta AND) por cada fila de la tabla de verdad para la que la función es igual a 1. Cada término producto contiene todas las variables de entrada, y se forma de tal modo que si la variable de entrada x_i es igual a 1 en la fila dada, entonces se introduce x_i en el término; si $x_i = 0$, entonces se introduce $\overline{x}_i$. La suma de estos términos producto realiza la función deseada. Segundo, existen muchos circuitos que pueden cumplir una función específica; algunos pueden ser más simples que otros. Mediante manipulación algebraica es posible derivar expresiones lógicas simplificadas y, por tanto, circuitos de menor costo.

El proceso mediante el cual comenzamos con una descripción del comportamiento funcional deseado y luego generamos un circuito que lo satisfaga se llama *síntesis*. En consecuencia, puede decirse que en la figura 2.16 se "sintetizaron" los circuitos a partir de la tabla de verdad de la figura 2.15. La generación de expresiones AND-OR a partir de una tabla de verdad sólo es uno de los muchos tipos de técnicas de síntesis que encontrará en esta obra.

2.6.1 FORMAS DE PRODUCTOS DE SUMAS Y SUMAS DE PRODUCTOS

Tras exponer el proceso de síntesis con un ejemplo muy simple, ahora lo presentaremos en términos más formales empleando la terminología de la bibliografía técnica. También mostraremos cómo aplicar el principio de dualidad, explicado en la sección 2.5, en el proceso de síntesis.

Si una función *f* se especifica en la forma de tabla de verdad, entonces es posible obtener una expresión que realice *f* considerando las filas de la tabla para las que $f = 1$, como ya se hizo, o las filas para las que $f = 0$, como veremos brevemente.

Mintérminos

Para una función de n variables, un término producto en el que cada una de las n variables aparezca una vez se llama *mintérmino*. Las variables pueden aparecer en un mintérmino en forma sin complementar o en complemento. Para una fila de la tabla de verdad, el mintérmino se forma incluyendo x_i si $x_i = 1$ y $\overline{x}_i$ si $x_i = 0$.

Para ilustrar este concepto considérese la tabla de verdad de la figura 2.17. Las filas están numeradas de 0 a 7, por lo que podemos hacer referencia a ellas con facilidad. (El lector familiarizado con la representación en números binarios notará que los números de fila elegidos son justo los representados por los patrones de bit de las variables x_1, x_2 y x_3; la representación numérica se abordará en el capítulo 5.) En la figura se muestran todos los mintérminos para la tabla de tres variables. Por ejemplo, en la primera fila las variables tienen los valores $x_1 = x_2 = x_3 = 0$, lo que conduce al mintérmino $\overline{x}_1\overline{x}_2\overline{x}_3$. En la segunda fila, $x_1 = x_2 = 0$ y $x_3 = 1$, lo cual produce el mintérmino $\overline{x}_1\overline{x}_2x_3$, etc. A fin de referir con facilidad mintérminos individuales conviene identificar cada uno de ellos mediante un índice que corresponda a los números de fila que se muestran en la figura. Emplearemos la notación m_i para denotar el mintérmino para el número de fila i. En consecuencia, $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$, $m_1 = \overline{x}_1\overline{x}_2x_3$, etcétera.

Forma de suma de productos

Una función f puede representarse mediante una expresión que sea una suma de mintérminos, en la que a cada uno de los mintérminos se le multiplica mediante la función AND con el valor de f para la valoración correspondiente de las variables de entrada. Por ejemplo, los mintérminos de dos variables son $m_0 = \overline{x}_1\overline{x}_2$, $m_1 = \overline{x}_1x_2$, $m_2 = x_1\overline{x}_2$, y $m_3 = x_1x_2$. La función de la figura 2.15 puede representarse como

$$\begin{aligned} f &= m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1 \\ &= m_0 + m_1 + m_3 \\ &= \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_1x_2 \end{aligned}$$

que es la forma derivada en la sección anterior por medio de un enfoque intuitivo. En la expresión resultante sólo aparecen los mintérminos correspondientes a las filas para las que $f = 1$.

Cualquier función f puede representarse mediante una suma de mintérminos que corresponda a las filas de la tabla de verdad para las que $f = 1$. La implementación resultante es funcional-

Número de fila	x_1	x_2	x_3	Mintérmino	Maxitérmino
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Figura 2.17 Mintérminos y maxitérminos de tres variables.

mente correcta y única, pero no siempre es la implementación de *f* más barata. Se dice que una expresión lógica que consta de términos producto (AND) que se suman (con OR) es la forma de *suma de productos* (SOP, *sum-of-products*). Si cada término producto es un mintérmino, entonces la expresión es la *suma canónica de productos* para la función *f*. Como vimos en el ejemplo de la figura 2.16, el primer paso del proceso de síntesis consiste en derivar una expresión en suma canónica de productos para la función dada. Luego esa expresión puede manipularse aplicando los teoremas y las propiedades vistos en la sección 2.5, a fin de hallar una expresión funcionalmente equivalente de suma de productos que sea más barata.

Veamos otro ejemplo. Considérese la función de tres variables $f(x_1, x_2, x_3)$, especificada mediante la tabla de verdad de la figura 2.18. Para sintetizar esta función hay que incluir los mintérminos m_1, m_4, m_5 y m_6. Al copiar éstos de la figura 2.17 se desemboca en la siguiente expresión de suma canónica de productos para *f*

$$f(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

Esta expresión puede manipularse como sigue

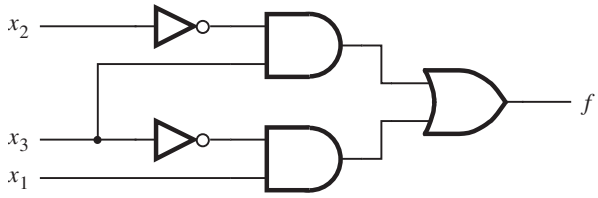
$$\begin{aligned} f &= (\overline{x}_1 + x_1)\overline{x}_2x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= 1 \cdot \overline{x}_2x_3 + x_1 \cdot 1 \cdot \overline{x}_3 \\ &= \overline{x}_2x_3 + x_1\overline{x}_3 \end{aligned}$$

Ésta es la expresión de suma de productos de costo mínimo para *f* y describe el circuito que se muestra en la figura 2.19*a*. Una buena indicación del costo de un circuito lógico es la cantidad total de compuertas más el número total de entradas a todas las compuertas en el circuito. Con esta medida el *costo* del circuito de la figura 2.19*a* es 13, ya que tiene cinco compuertas y ocho entradas a ellas. En comparación, el circuito implementado con base en la suma canónica de productos tendría un costo de 27; de la expresión precedente, la compuerta OR tiene cuatro entradas, cada una de las cuatro compuertas AND tiene tres entradas y cada una de las tres compuertas NOT tiene una entrada.

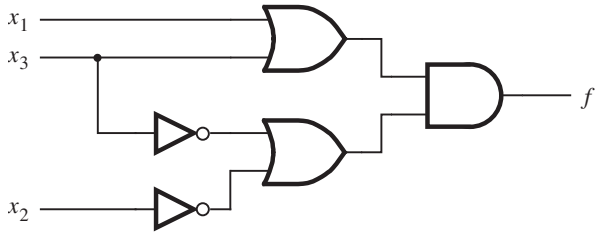
Los mintérminos, con sus subíndices correspondientes al número de fila, también pueden usarse para especificar una función de una manera más concisa.

Número de fila	x_1	x_2	x_3	$f(x_1\ x_2\ x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figura 2.18 Función de tres variables.



a) Realización mínima de suma de productos



b) Realización mínima de productos de sumas

Figura 2.19 Dos realizaciones de la función de la figura 2.18.

Por ejemplo, la función de la figura 2.18 puede especificarse como

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

o incluso de manera más sencilla como

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

El signo $\sum$ denota la operación suma lógica. Esta notación taquigráfica suele utilizarse en la práctica.

Maxitérminos

El principio de dualidad indica que si es posible sintetizar una función f considerando las filas de la tabla de verdad para las que $f = 1$, entonces también debe ser posible sintetizar f considerando las filas para las que $f = 0$. Este enfoque alternativo usa los complementos de los mintérminos, llamados *maxitérminos*. En la figura 2.17 se presenta una lista de todos los maxitérminos posibles para las funciones de tres variables. Haremos referencia a un maxitérmino M_j mediante el mismo número de fila que su correspondiente mintérmino m_j, como se observa en la figura.

Forma de producto de sumas

Si una función f se especifica mediante una tabla de verdad, entonces su complemento $\overline{f}$ puede representarse con una suma de mintérminos para los que $\overline{f} = 1$, que son las filas donde

$f = 0$. Por ejemplo, para la función de la figura 2.15

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2 \\ &= x_1\overline{x}_2\end{aligned}$$

Si esta expresión se complementa con el teorema de DeMorgan, el resultado es

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x}_2} \\ &= \overline{x}_1 + x_2\end{aligned}$$

Nótese que esta expresión ya se obtuvo antes mediante la manipulación algebraica de la forma en suma canónica de productos para la función *f*. El punto clave aquí es que

$$f = \overline{m}_2 = M_2$$

donde M_2 es el maxitérmino para la fila 2 de la tabla de verdad.

Veamos otro ejemplo. Considérese de nuevo la función de la figura 2.18, cuyo complemento puede representarse como

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7 \\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

Luego, *f* se expresa como

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7} \\ &= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7 \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)\end{aligned}$$

Esta expresión representa *f* como un producto de maxitérminos.

Se dice que una expresión lógica que consta de términos suma (OR) que son los factores de un producto lógico (AND) es la forma en *producto de sumas* (POS, *product-of-sums*). Si cada término suma es un maxitérmino, entonces la expresión se denomina *producto canónico de sumas* para la función dada. Cualquier función *f* puede sintetizarse encontrando su producto canónico de sumas. Ello supone tomar el maxitérmino de cada fila de la tabla de verdad para el que $f = 0$ y formar un producto de estos maxitérminos.

Volvamos al ejemplo anterior. Es posible reducir la complejidad de la expresión derivada que comprende un producto de maxitérminos. Si usamos las propiedades conmutativa, 10*b*, y asociativa, 11*b*, de la sección 2.5 podemos escribir esta expresión como

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(x_1 + (\overline{x}_2 + \overline{x}_3))(\overline{x}_1 + (\overline{x}_2 + \overline{x}_3))$$

Luego, con la propiedad de combinación, 14*b*, la expresión se reduce a

$$f = (x_1 + x_3)(\overline{x}_2 + \overline{x}_3)$$

El circuito correspondiente aparece en la figura 2.19*b*. Su costo es 13. Aunque resulta ser el mismo que el de la versión en suma de productos de la figura 2.19*a*, el lector no debe suponer que

el costo de un circuito derivado en la forma de suma de productos en general será igual al de un circuito correspondiente derivado en la forma de producto de sumas.

Si se utiliza la notación abreviada, una forma alternativa de especificar nuestra función de ejemplo es

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

o de manera más simple

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

El signo Π denota la operación producto lógico.

En la explicación anterior se mostró cómo realizar las funciones lógicas en la forma de circuitos lógicos, que constan de circuitos de compuertas que implementan funciones básicas. Es posible realizar una función específica con circuitos de una estructura distinta, lo que casi siempre supone una diferencia en costo. Un objetivo importante para un diseñador es minimizar el costo del circuito diseñado. En el capítulo 4 estudiaremos las técnicas más relevantes para encontrar implementaciones de costo mínimo.

Ejemplo 2.3 Considérese la función

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

La expresión SOP canónica para la función se deriva mediante mintérminos

$$\begin{aligned} f &= m_2 + m_3 + m_4 + m_6 + m_7 \\ &= \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3 \end{aligned}$$

Esta expresión puede simplificarse usando las identidades de la sección 2.5 como sigue

$$\begin{aligned} f &= \overline{x}_1 x_2 (\overline{x}_3 + x_3) + x_1 (\overline{x}_2 + x_2) \overline{x}_3 + x_1 x_2 (\overline{x}_3 + x_3) \\ &= \overline{x}_1 x_2 + x_1 \overline{x}_3 + x_1 x_2 \\ &= (\overline{x}_1 + x_1) x_2 + x_1 \overline{x}_3 \\ &= x_2 + x_1 \overline{x}_3 \end{aligned}$$

Ejemplo 2.4 Considérese de nuevo la función del ejemplo 2.3. En lugar de usar los mintérminos, podemos especificar esta función como un producto de maxitérminos para los que $f = 0$

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 5)$$

Entonces, la expresión POS canónica se deriva como

$$\begin{aligned} f &= M_0 \cdot M_1 \cdot M_5 \\ &= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3) \end{aligned}$$

Una expresión POS simplificada puede derivarse como

$$\begin{aligned} f &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \overline{x}_3)(x_1 + (x_2 + \overline{x}_3))(\overline{x}_1 + (x_2 + \overline{x}_3)) \\ &= ((x_1 + x_2) + x_3\overline{x}_3)(x_1\overline{x}_1 + (x_2 + \overline{x}_3)) \\ &= (x_1 + x_2)(x_2 + \overline{x}_3) \end{aligned}$$

Nótese que con la propiedad distributiva, 12*b*, esta expresión conduce a

$$f = x_2 + x_1\overline{x}_3$$

que es la misma que la expresión derivada en el ejemplo 2.3.

Ejemplo 2.5

Supóngase que una función de cuatro variables se define mediante

$$f(x_1, x_2, x_3, x_4) = \sum m(3, 7, 9, 12, 13, 14, 15)$$

La expresión SOP canónica para esta función es

$$f = \overline{x}_1\overline{x}_2x_3x_4 + \overline{x}_1x_2x_3x_4 + x_1\overline{x}_2\overline{x}_3x_4 + x_1x_2\overline{x}_3\overline{x}_4 + x_1x_2\overline{x}_3x_4 + x_1x_2x_3\overline{x}_4 + x_1x_2x_3x_4$$

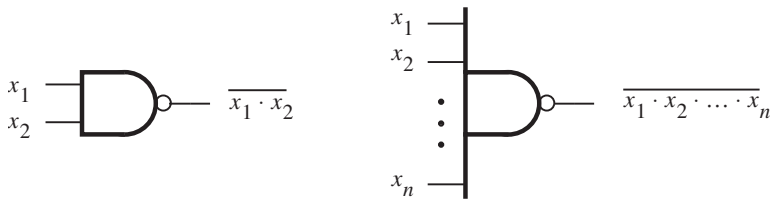
Puede obtenerse una expresión SOP más simple del modo siguiente

$$\begin{aligned} f &= \overline{x}_1(\overline{x}_2 + x_2)x_3x_4 + x_1(\overline{x}_2 + x_2)\overline{x}_3x_4 + x_1x_2\overline{x}_3(\overline{x}_4 + x_4) + x_1x_2x_3(\overline{x}_4 + x_4) \\ &= \overline{x}_1x_3x_4 + x_1\overline{x}_3x_4 + x_1x_2\overline{x}_3 + x_1x_2x_3 \\ &= \overline{x}_1x_3x_4 + x_1\overline{x}_3x_4 + x_1x_2(\overline{x}_3 + x_3) \\ &= \overline{x}_1x_3x_4 + x_1\overline{x}_3x_4 + x_1x_2 \end{aligned}$$

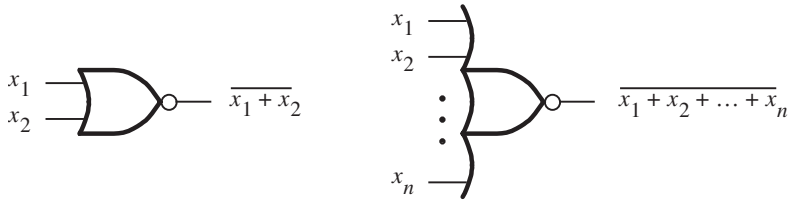
2.7 CIRCUITOS LÓGICOS NAND Y NOR

Ya explicamos el uso de las compuertas AND, OR y NOT en la síntesis de circuitos lógicos. Hay otras funciones lógicas básicas que sirven para el mismo propósito. Muy útiles son las funciones NAND y NOR, que se obtienen al complementar la salida generada por las operaciones AND y OR, respectivamente, y cuyo atractivo radica en que pueden implementarse con circuitos electrónicos más simples que las funciones AND y OR, como veremos en el capítulo siguiente. En la figura 2.20 se presentan los símbolos gráficos de las compuertas NAND y NOR. A fin de representar la señal de salida complementada se coloca un pequeño círculo en el lado de la salida correspondiente.

Si se realizan las compuertas NAND y NOR con circuitos más sencillos que las compuertas AND y OR, entonces cabe preguntar si pueden usarse de modo directo en la síntesis de circuitos lógicos. En la sección 2.5 expusimos el teorema de DeMorgan. En la figura 2.21 se muestra su interpretación en compuerta lógica. En el inciso *a*) de la figura se interpreta la identidad 15*a*. En ella se especifica que una función NAND de variables x_1 y x_2 equivale a complementar primero cada una de las variables y luego a aplicarles la función OR. Nótese que en el lado derecho hemos indicado las compuertas NOT simplemente como pequeños círculos, lo que denota

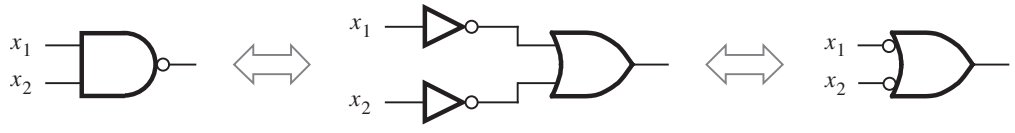


a) Compuertas NAND

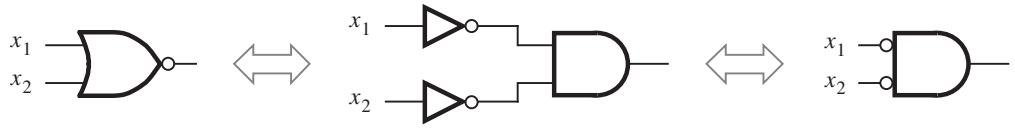


b) Compuertas NOR

Figura 2.20 Compuertas NAND y NOR.



a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Figura 2.21 Teorema de DeMorgan en términos de compuertas lógicas.

inversión del valor lógico en ese punto. La otra mitad del teorema de DeMorgan, la identidad 15*b*, aparece en el inciso *b*) de la figura, donde se afirma que la función NOR equivale a invertir primero las variables de entrada y luego a operarlas con la función AND.

En la sección 2.6 explicamos cómo implementar cualquier función lógica en forma de suma de productos o en forma de producto de sumas, lo que lleva a circuitos lógicos que tienen una estructura AND-OR o OR-AND respectivamente. Ahora demostraremos que tales circuitos pueden implementarse sólo con compuertas NAND o sólo con compuertas NOR.

Considérese el circuito de la figura 2.22 como representativo de los circuitos generales AND-OR. Es posible transformar este circuito en uno de compuertas NAND, como se advierte en la figura. Primero, se sustituye cada conexión entre la compuerta AND y la compuerta OR con una conexión que incluya dos inversiones de la señal: una en la salida de la compuerta AND y otra en la entrada de la compuerta OR. Esta doble inversión no tiene efecto en el comportamiento del circuito, como se afirmó formalmente en el teorema 9 de la sección 2.5. De acuerdo con la figura 2.21*a*, la compuerta OR con inversiones en sus entradas equivale a una compuerta NAND. Por tanto, el circuito puede volverse a dibujar usando únicamente compuertas NAND, como se muestra en la figura 2.22. Este ejemplo indica que es posible implementar cualquier circuito AND-OR como un circuito NAND-NAND que tenga la misma topología.

En la figura 2.23 se presenta una construcción similar para un circuito de producto de sumas, que puede transformarse en un circuito sólo con compuertas NOR. El procedimiento es exactamente el mismo que el descrito para la figura 2.22, excepto que ahora se aplica la identidad de la figura 2.21*b*. La conclusión es que cualquier circuito OR-AND puede implementarse como un circuito NOR-NOR que tenga la misma topología.

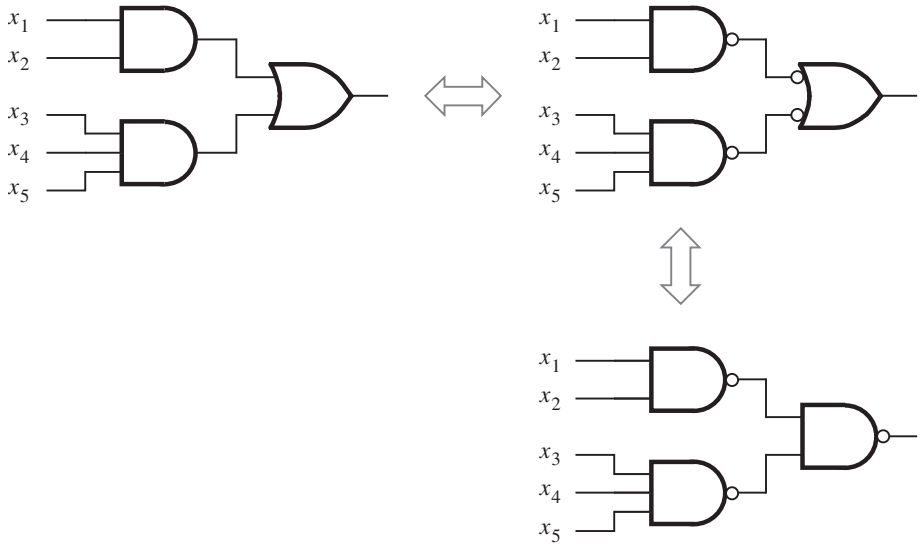


Figura 2.22 Uso de compuertas NAND para implementar una suma de productos.

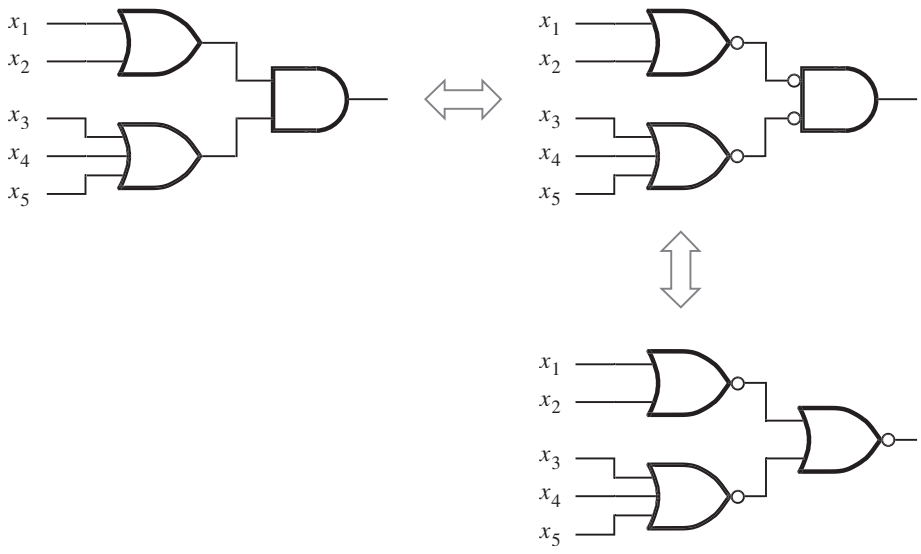


Figura 2.23 Uso de compuerta NOR para implementar un producto de sumas.

Ejemplo 2.6 Implemente la función

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

empleando sólo compuertas NOR. En el ejemplo 2.4 se demostró que la función puede representarse mediante la expresión POS

$$f = (x_1 + x_2)(x_2 + \overline{x}_3)$$

En la figura 2.24*a* se observa un circuito OR-AND que corresponde a esta expresión. Con la misma estructura del circuito, la figura 2.24*b* muestra una versión en compuerta NOR. Nótese que x_3 se invierte mediante una compuerta NOR que tiene sus entradas unidas.

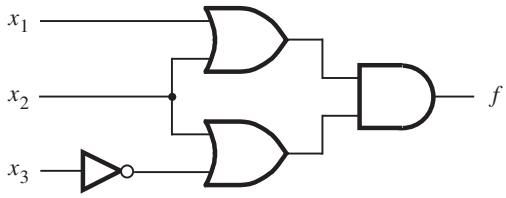
Ejemplo 2.7 Ahora implemente la función

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

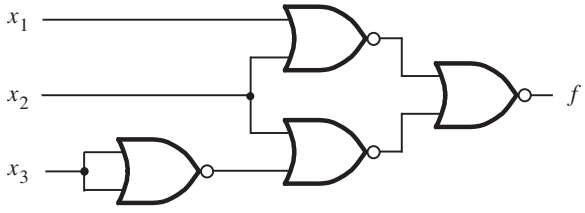
utilizando solamente compuertas NAND. En el ejemplo 2.3 derivamos la expresión SOP

$$f = x_2 + x_1\overline{x}_3$$

que se realizó por medio del circuito presentado en la figura 2.25*a*. De nuevo podemos usar la misma estructura para obtener un circuito con compuertas NAND, pero con una diferencia. La señal x_2 sólo pasa por una compuerta OR, en vez de hacerlo por una compuerta AND y una compuerta OR. Si nada más sustituimos la compuerta OR con una compuerta NAND, esta señal se invertirá, lo que resultará en un valor de salida equivocado. En virtud de que x_2 debe no invertirse

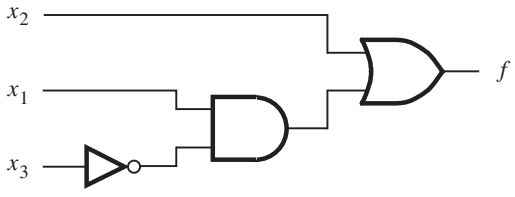


a) Implementación POS

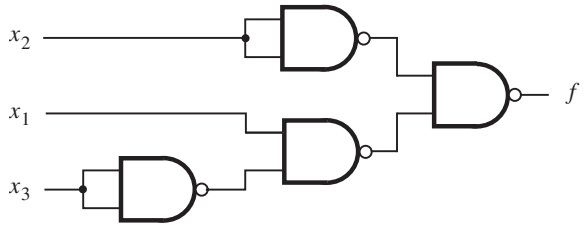


b) Implementación NOR

Figura 2.24 Realización con compuertas NOR de la función del ejemplo 2.4.



a) Implementación SOP



b) Implementación NAND

Figura 2.25 Realización con compuertas NAND de la función del ejemplo 2.3.

o invertirse dos veces, se puede pasar por dos compuertas NAND, como se muestra en la figura 2.25*b*. Obsérvese que para este circuito la salida f es

$$f = \overline{\overline{x}_2 \cdot \overline{x_1 \overline{x}_3}}$$

Al aplicar el teorema de DeMorgan esta expresión se convierte en

$$f = x_2 + x_1 \overline{x}_3$$

2.8 EJEMPLOS DE DISEÑO

Los circuitos lógicos ofrecen una solución a un problema. Implementan funciones necesarias para llevar a cabo tareas específicas. En el marco de las computadoras, los circuitos lógicos brindan plena capacidad para la ejecución de programas y el procesamiento de datos. Tales circuitos son complejos y difíciles de diseñar. Pero sin importar su complejidad, un diseñador de circuitos lógicos siempre encara el mismo conflicto esencial. Primero ha de especificar el comportamiento deseado del circuito. Después debe sintetizarlo e implementarlo. Por último, debe probarlo a fin de verificar que cumple las especificaciones. El comportamiento deseado inicialmente se describe con palabras, que luego han de convertirse en una especificación formal. En esta sección daremos dos ejemplos sencillos de diseño.

2.8.1 CONTROL DE LUZ DE TRES VÍAS

Supóngase que una habitación grande tiene tres puertas y que un interruptor cerca de cada una de ellas controla la luz. Debe ser posible encenderla y apagarla mediante el cambio de estado de cualquiera de los interruptores.

Primero convirtamos este enunciado en palabras en una especificación formal por medio de una tabla de verdad. Sean x_1, x_2 y x_3 las variables de entrada que denotan el estado de cada interruptor. Supóngase que la luz está apagada si todos los interruptores están abiertos. Si se cierra alguno de ellos, la luz se enciende. Luego, la luz se apaga si se acciona otro. Por tanto, la luz se encenderá exactamente si un interruptor se cierra, y se apagará si dos (o ninguno) interruptores se cierran. Si la luz está apagada cuando dos interruptores están cerrados, entonces debe ser posible encenderla cerrando el tercer interruptor. Si $f(x_1, x_2, x_3)$ representa el estado de la luz, el comportamiento funcional requerido se especifica como se muestra en la tabla de verdad de la figura 2.26. La expresión en suma canónica de productos de la función especificada es

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x}_1 \overline{x}_2 x_3 + \overline{x}_1 x_2 \overline{x}_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 x_3 \end{aligned}$$

Esta expresión no puede simplificarse en una expresión de suma de productos de menor costo. En la figura 2.27*a* se presenta el circuito resultante.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figura 2.26 Tabla de verdad para el control de luz de tres vías.

Una realización alternativa de esta función se halla en la forma de producto de sumas, cuya expresión canónica es

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3) \end{aligned}$$

El circuito resultante se ilustra en la figura 2.27*b*. Cuesta lo mismo que el circuito del inciso *a*) de la figura.

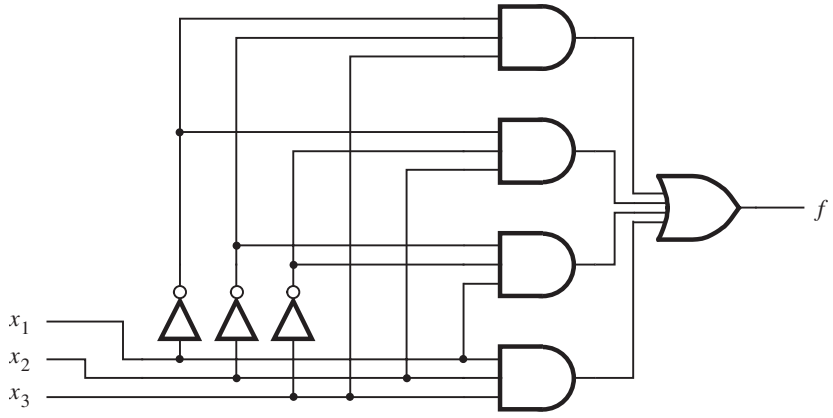
Cuando se implementa el circuito diseñado puede probarse aplicando varias valoraciones de entrada al circuito y comprobando si la salida corresponde a los valores especificados en la tabla de verdad. Un método directo consiste en comprobar que produce la salida correcta para los ocho posibles valores de entrada.

2.8.2 CIRCUITO MULTIPLEXOR

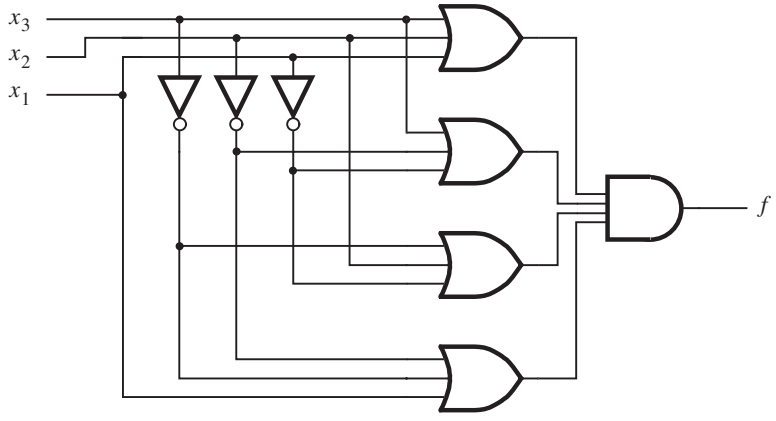
En los sistemas de cómputo a menudo es necesario elegir datos provenientes exactamente de una de varias fuentes posibles. Supóngase que existen dos fuentes de datos que proporcionan como señales de entrada x_1 y x_2. Los valores de estas señales cambian con el tiempo, quizás a intervalos regulares. Por ende, en cada una de las entradas x_1 y x_2 se aplican secuencias de 0 y 1. Queremos diseñar un circuito que produzca una salida que tenga el mismo valor que x_1 o x_2, que dependa del valor de una señal de control de selección s. Por tanto, el circuito debe tener tres entradas: x_1, x_2 y s. Digamos que la salida del circuito será la misma que el valor de entrada x_1 si $s = 0$, y será la misma que x_2 si $s = 1$.

Con base en estos requisitos podemos especificar el circuito deseado en la forma de una tabla de verdad, la de la figura 2.28*a*. A partir de ella se deriva la suma canónica de productos

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x}_2 + \overline{s}x_1x_2 + s\overline{x}_1x_2 + sx_1x_2$$



a) Realización en suma de productos



b) Realización en producto de sumas

Figura 2.27 Implementación de la función de la figura 2.26.

Si se emplea la propiedad distributiva esta expresión puede escribirse como

$$f = \overline{s}x_1(\overline{x}_2 + x_2) + s(\overline{x}_1 + x_1)x_2$$

La aplicación del teorema 8*b* produce

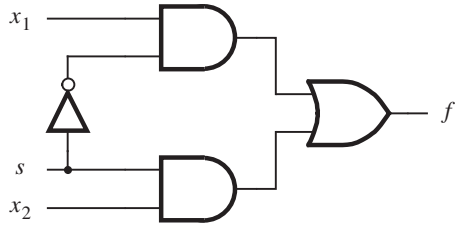
$$f = \overline{s}x_1 \cdot 1 + s \cdot 1 \cdot x_2$$

Finalmente, con el teorema 6*a* se obtiene

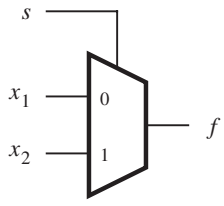
$$f = \overline{s}x_1 + sx_2$$

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

a) Tabla de verdad



b) Circuito



c) Símbolo gráfico

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

d) Representación más compacta de la tabla de verdad

Figura 2.28 Implementación de un multiplexor.

En la figura 2.28*b* se muestra un circuito que implementa esta función. Los circuitos de este tipo se usan tan ampliamente que reciben un nombre especial. Un circuito que genera una salida que refleja con exactitud el estado de una de varias entradas de datos, con base en el valor de una o más entradas de control de selección, se llama *multiplexor*. Se dice que un circuito multiplexor "multiplexa" las señales de entrada en una sola salida.

En este ejemplo derivamos un multiplexor con dos entradas de datos, el cual se denomina "multiplexor 2 a 1". En la figura 2.28*c* se muestra un símbolo gráfico usado comúnmente para el multiplexor 2 a 1. La misma idea puede extenderse a circuitos más grandes. Un multiplexor 4 a 1 tiene cuatro entradas de datos y una salida. En este caso se necesitan dos entradas de control de selección para elegir una de las cuatro entradas de datos transmitidos como la señal de salida. Un multiplexor 8 a 1 precisa ocho entradas de datos y tres entradas de control de selección, etcétera.

Nótese que el enunciado "$f = x_1$ si $s = 0$, y $f = x_2$ si $s = 1$" puede presentarse en una forma más compacta de tabla de verdad, como se indica en la figura 2.28*d*. En capítulos posteriores tendremos oportunidad de usar tal representación.

Se mostró cómo es posible construir un multiplexor con compuertas AND, OR y NOT. Puede usarse la misma estructura de circuito para implementar el multiplexor con compuertas NAND, como se explica en la sección 2.7. En el capítulo 3 estudiaremos otras posibilidades para construir multiplexores, y en el 6 analizaremos de manera pormenorizada el uso de éstos.

Los diseñadores de circuitos lógicos se apoyan enormemente en las herramientas CAD. En esta obra se busca alentar al lector a conocer cuanto antes la herramienta CAD ofrecida en el libro. En este punto es útil dar una introducción a estas herramientas. En la sección siguiente se presentan algunos conceptos básicos necesarios para usarlas. Asimismo, en la sección 2.10 se expone un lenguaje especial para describir circuitos lógicos llamado VHDL, que se emplea para describir los circuitos como una entrada para las herramientas CAD, que entonces proceden a derivar una implementación adecuada.

2.9 INTRODUCCIÓN A LAS HERRAMIENTAS CAD

En las secciones precedentes se presentó un método básico para sintetizar circuitos lógicos. Un diseñador podría aplicarlo manualmente para circuitos pequeños. Sin embargo, los circuitos lógicos que se encuentran en sistemas complejos, como las computadoras actuales, no pueden diseñarse a mano, es preciso hacerlo con modernas herramientas CAD que implementan de forma automática las técnicas de síntesis.

Para diseñar un circuito lógico se requieren varias herramientas CAD. Casi siempre están empaquetadas en un *sistema CAD*, que por lo general incluye herramientas para las tareas siguientes: ingreso del diseño, síntesis y optimación, simulación y diseño físico. Estudiaremos algunas de estas herramientas en la presente sección y en capítulos posteriores ahondaremos en ello.

2.9.1 INGRESO DEL DISEÑO

El punto de partida en el proceso de diseñar un circuito lógico es la concepción de lo que se supone debe hacer éste y el planteamiento de su estructura general. Este paso lo efectúa manualmente el diseñador, pues se requiere experiencia de diseño e intuición. El resto del proceso de diseño se realiza con el auxilio de las herramientas CAD. La primera etapa de este proceso supone ingresar en el sistema CAD una descripción del circuito que se va a diseñar. Esta etapa se denomina *ingreso del diseño*. Describiremos dos métodos de ingreso de diseño: el uso de captura esquemática y la escritura de código fuente en un lenguaje de descripción de hardware.

Captura esquemática

Un circuito lógico puede describirse dibujando las compuertas lógicas e interconectándolas con cables. La herramienta CAD para ingresar el diseño de un circuito de esta manera se llama *herramienta de captura esquemática*. La palabra *esquemática* se refiere al diagrama de un circuito en el que los elementos de éste, como las compuertas lógicas, se muestran como símbolos gráficos y las conexiones entre tales elementos se indican con líneas.

Una herramienta de captura esquemática usa las funciones gráficas de una computadora; por su parte, el ratón permite al usuario trazar un diagrama esquemático. Para facilitar la inclusión de compuertas en el esquema la herramienta ofrece un juego de símbolos gráficos que representan compuertas de varios tipos con diferentes números de entradas. Este juego de símbolos se llama *biblioteca*. Las compuertas de la biblioteca pueden importarse al esquema del usuario, y la herramienta brinda una forma gráfica de interconectarlas para crear un circuito lógico.

Es posible representar cualesquiera subcircuitos creados anteriormente como símbolos gráficos e incluirse en el esquema. En la práctica es común que el usuario de un sistema CAD cree un circuito que comprenda otros circuitos más pequeños. Este método se conoce como *diseño jerárquico* y provee una buena forma de manejar la complejidad propia de los circuitos grandes.

La herramienta de captura esquemática se describe con detalle en el apéndice B. Aunque es simple de usar, se vuelve engorrosa cuando enfrenta circuitos grandes. Un mejor método para abordar éstos es escribir código fuente mediante un lenguaje de descripción de hardware para representar el circuito.

Lenguajes de descripción de hardware

Un *lenguaje de descripción de hardware* (HDL, *hardware description language*) es similar a un lenguaje de programación típico, salvo que el HDL sirve para describir hardware en lugar de un programa que la computadora ejecutará. Hay muchos HDL comerciales. Algunos son sujetos a un derecho de propiedad, lo que significa que los ofrece una compañía y sólo pueden usarse para implementar circuitos en la tecnología que esa compañía ofrece. En este libro no analizaremos los HDL con derechos de propiedad. En vez de ello nos centraremos en un lenguaje que apoyan prácticamente todos los comercios que ofrecen tecnología de hardware digital y oficialmente se respalda como una norma del *Instituto de Ingenieros Eléctricos* y *Electrónicos* (IEEE, *Institute of Electrical and Electronics Engineers*). El IEEE es un organismo mundial que promueve actividades técnicas para el beneficio de la sociedad. Una de ellas supone el desarrollo de normas que definan cómo usar ciertos conceptos tecnológicos de modo adecuado para un gran grupo de usuarios.

Dos HDL son normas del IEEE: VHDL (*Very High Speed Integrated Circuit Hardware Description Language*: lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad) y *Verilog VHDL*. Ambos lenguajes tienen amplio uso en la industria. En esta obra emplearemos VHDL, pero la editorial [4] tiene a disposición una versión en Verilog de lo expuesto en el libro. Aunque los dos lenguajes difieren en mucho, la elección de usar uno u otro cuando estudie circuitos lógicos no reviste especial importancia porque ambos ofrecen características similares. Los conceptos ilustrados en el libro mediante VHDL pueden aplicarse directamente en Verilog.

En comparación con realizar captura esquemática, el uso de VHDL da varias ventajas. Puesto que lo apoyan la mayor parte de los organismos que ofrecen tecnología de hardware digital, VHDL brinda *portabilidad* de diseño. Un circuito especificado en VHDL puede implementarse en diferentes tipos de chips y con herramientas CAD ofrecidas por diferentes compañías, sin

necesidad de cambiar la especificación en VHDL. La portabilidad de diseño es una ventaja importante porque la tecnología de circuitos digitales cambia con rapidez. Al utilizar un lenguaje estándar el diseñador puede centrarse en la funcionalidad del circuito deseado sin preocuparse mucho por los detalles de la tecnología que a la postre usará para la implementación.

El ingreso de diseño de un circuito lógico se efectúa mediante la escritura de código VHDL. Las señales del circuito pueden representarse como variables en el código fuente, y las funciones lógicas se expresan mediante la asignación de valores a dichas variables. El código fuente de VHDL es texto llano, lo que facilita al diseñador incluir en él la documentación que explique cómo funciona el circuito. Esta característica, aunada al hecho de que VHDL se usa ampliamente, alienta a compartir y reutilizar los circuitos descritos en VHDL. Esto permite el desarrollo más rápido de productos nuevos en casos donde el código VHDL existente puede adaptarse para diseñar circuitos nuevos.

Similar al modo en que los circuitos grandes se manejan en la captura esquemática, el código VHDL puede escribirse en forma modular que facilite el diseño jerárquico. El diseño de circuitos lógicos pequeños y grandes pueden representarse eficientemente en código VHDL. Este lenguaje se usa para definir circuitos como microprocesadores con millones de transistores.

El ingreso de diseño en VHDL puede combinarse con otros métodos. Por ejemplo, es posible usar una herramienta de captura esquemática en la que un subcircuito del esquema se describa con VHDL. En la sección 2.10 se estudiará más de VHDL.

2.9.2 SÍNTESIS

La síntesis es el proceso por el que se genera un circuito lógico a partir de una especificación inicial que puede proporcionarse en forma de diagrama esquemático o de código escrito en un lenguaje de descripción de hardware. Con base en esa especificación las herramientas CAD de síntesis generan implementaciones eficientes de circuitos.

El proceso de traducción, o *compilación*, del código de VHDL en un circuito de compuertas lógicas forma parte de la síntesis. La salida es un conjunto de expresiones lógicas que describen las funciones lógicas necesarias para realizar el circuito.

Sin importar el tipo de ingreso de diseño que se use, las expresiones lógicas iniciales producidas por las herramientas de síntesis no tendrán una forma óptima, ya que reflejan lo que el diseñador ingresa en las herramientas CAD. Es imposible que un diseñador produzca manualmente resultados óptimos para circuitos grandes. Por ende, una de las tareas importantes de las herramientas de síntesis es manipular el diseño del usuario a fin de generar de manera automática un circuito equivalente, pero mejor.

La medida de lo que hace a un circuito mejor que otro depende tanto de las necesidades particulares de un proyecto de diseño como de la tecnología elegida para la implementación. En la sección 2.6 señalamos que un buen circuito puede ser el que tenga el menor costo. Hay otras posibles metas de optimación, motivadas por el tipo de tecnología de hardware usado para implementar el circuito. En el capítulo 3 estudiaremos las tecnologías de implementación y en el 4 volveremos al tema de las metas de optimación.

El desempeño de un circuito sintetizado puede evaluarse construyendo y probando físicamente el circuito, pero también mediante la simulación.

2.9.3 SIMULACIÓN FUNCIONAL

Un circuito representado en forma de expresiones lógicas se simula, entre otras cosas, para verificar que funcionará como se espera. La herramienta que cumple esta tarea recibe el nombre de *simulador funcional*. Utiliza las expresiones lógicas (conocidas como *ecuaciones*) generadas durante la síntesis y supone que se implementarán con compuertas perfectas por las que pasarán instantáneamente las señales. El simulador requiere que el usuario especifique las valoraciones de las entradas del circuito que han de aplicarse durante la simulación. Para cada una de ellas el simulador evalúa las salidas producida por las expresiones. Los resultados de la simulación suelen entregarse en forma de diagrama de tiempo que el usuario examina para verificar que el circuito opera como se requiere. La simulación funcional es un tema que se aborda con mayor hondura en el apéndice B.

2.9.4 DISEÑO FÍSICO

Después de la síntesis lógica el paso siguiente en el flujo de diseño consiste en determinar con exactitud cómo implementar el circuito en un chip. Este paso se denomina *diseño físico*. Como veremos en el próximo capítulo, hay varias tecnologías para implementar los circuitos lógicos. Las herramientas de diseño físico mapean un circuito especificado mediante expresiones lógicas en una realización que utiliza los recursos disponibles en el chip. Ello determina la ubicación de los elementos lógicos específicos, que no necesariamente son simples compuertas de los tipos expuestos hasta ahora. También establece las conexiones de cable que deben llevarse a cabo entre tales elementos para construir el circuito deseado.

2.9.5 SIMULACIÓN DE TIEMPO

Tanto las compuertas como otros elementos lógicos se implementan con circuitos electrónicos, como veremos en el capítulo 3. Un circuito electrónico no puede cumplir su función de manera instantánea. Cuando cambian los valores de las entradas al circuito se precisa cierto tiempo antes que ocurra el cambio correspondiente en la salida. Esto se llama *retardo de propagación* del circuito. El retardo de propagación consta de dos tipos de retardo. Cada elemento lógico necesita cierto lapso para generar una señal de salida válida siempre que haya cambios en los valores de sus entradas. Aparte de este retardo, existe un retardo producido por las señales que deben propagarse por los cables que conectan los diversos elementos lógicos. El efecto combinado es que los circuitos reales muestran retardos, lo que tiene un efecto significativo en su rapidez de operación.

Un *simulador de tiempo* evalúa los retardos esperados del circuito lógico diseñado. Su resultado sirve para determinar si éste satisface los requisitos de tiempo de la especificación para el diseño. Si no es así, el diseñador puede solicitar que las herramientas de diseño físico lo intenten de nuevo indicando restricciones temporales específicas que han de satisfacerse. Si esto no resulta, entonces el diseñador debe probar diferentes optimaciones en el paso de síntesis, o bien mejorar el diseño inicial presentado a las herramientas de síntesis.

2.9.6 CONFIGURACIÓN DE CHIP

Tras cerciorarse de que el circuito diseñado satisface todos los requisitos de la especificación, se implementa en un chip real. Este paso se llama *configuración* o *programación de chip*.

Las herramientas CAD analizadas en esta sección son las partes esenciales de un sistema CAD. En la figura 2.29 se ilustra todo el flujo de diseño expuesto. La presente es sólo una breve introducción. En el capítulo 12 se brinda una exposición completa de las herramientas CAD.

En este punto el lector debe haberse formado una idea de lo que supone el uso de herramientas CAD. Sin embargo, éstas sólo pueden apreciarse por completo cuando se usan de primera mano. Los apéndices B a D contienen tutoriales paso a paso que ilustran cómo usar el sistema CAD Quartus II incluido en el disco compacto que acompaña a esta obra. Se recomienda al lector que trabaje con el material de práctica presentado en esos apéndices. Puesto que los tutoriales emplean el lenguaje VHDL para el ingreso de diseño, en la sección siguiente se presenta una introducción a él.

2.10 INTRODUCCIÓN A VHDL

En el decenio de 1980, los rápidos avances en la tecnología de los circuitos integrados impulsaron el desarrollo de prácticas estándar de diseño para los circuitos digitales. VHDL se creó como parte de tal esfuerzo y se convirtió en el lenguaje estándar industrial para describir circuitos digitales, principalmente porque es un estándar oficial de la IEEE. En 1987 se adoptó la norma original para VHDL, llamada *IEEE* 1076. En 1993 se adoptó una norma revisada, la IEEE 1164.

En sus orígenes, VHDL tenía dos propósitos centrales. Primero, servía como lenguaje de documentación para describir la estructura de circuitos digitales complejos. Como estándar oficial del IEEE, ofreció una forma común de documentar los circuitos diseñados por varias personas. Segundo, VHDL aportó funciones para modelar el comportamiento de un circuito digital, lo que permitió emplearlo como entrada para programas que entonces se usaban para simular la operación del circuito.

En años recientes, aparte de usarlo para documentación y simulación, VHDL también se volvió popular para el ingreso de diseño en sistemas CAD. Las herramientas CAD se utilizan para sintetizar el código de VHDL en una implementación de hardware del circuito descrito. En este libro utilizaremos principalmente VHDL para la síntesis.

VHDL es un lenguaje complejo y refinado. Aunque aprender todas sus funciones es una tarea atemorizante, para usarlo en la síntesis basta conocer un subconjunto de ellas. Para simplificar la exposición nos centraremos en las características de VHDL que realmente se usan en los ejemplos del libro. Lo presentado debe ser suficiente para que el lector diseñe un repertorio amplio de circuitos. Quien desee aprender VHDL por completo puede remitirse a uno de los textos especializados [5-10].

VHDL se explica en varias etapas a lo largo de la obra. Nuestro enfoque general consiste en introducir funciones específicas sólo cuando sean relevantes para los temas de diseño que se aborden en la parte del texto respectiva. En el apéndice A se proporciona un resumen conciso de las funciones de VHDL expuestas en el libro. El lector encontrará conveniente remitirse a dicho

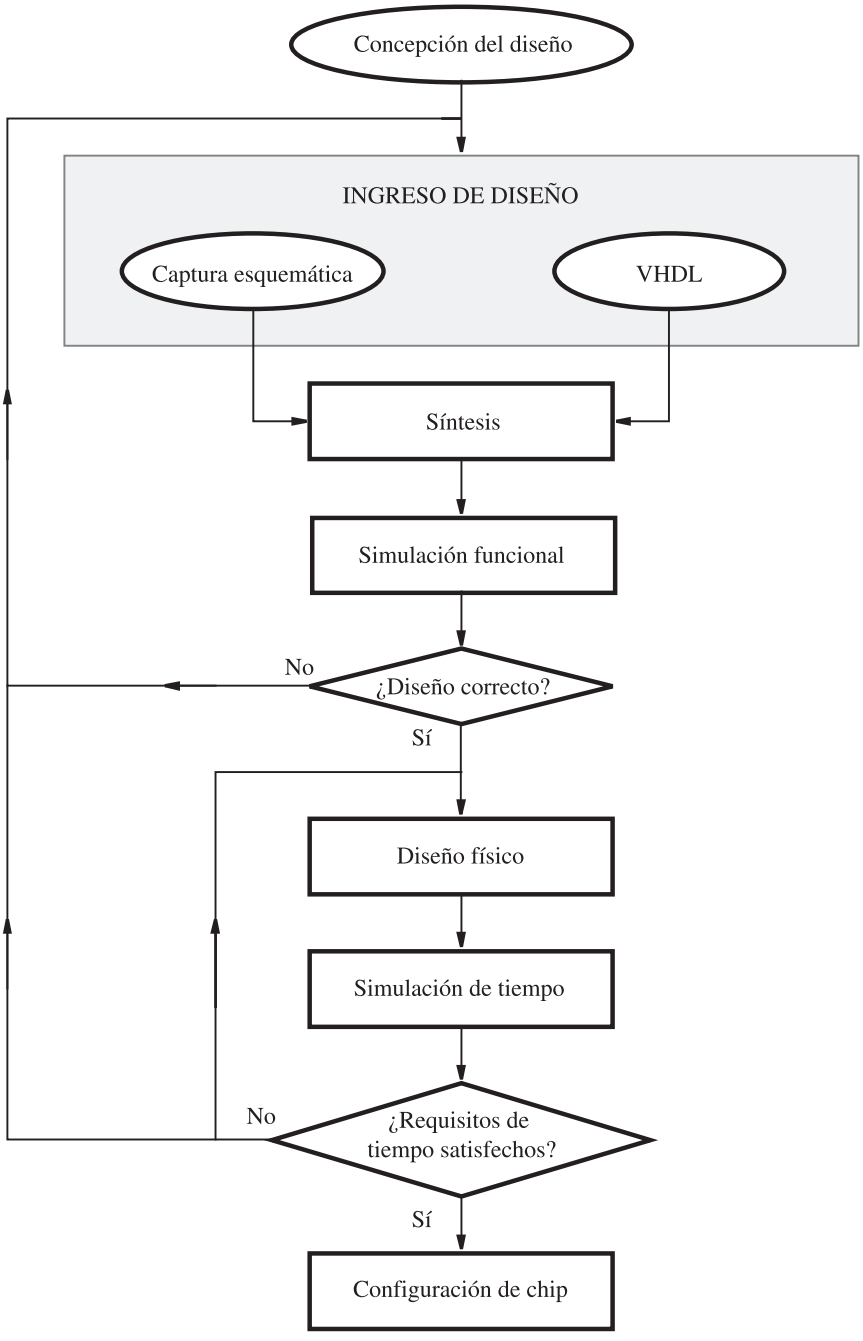


Figura 2.29 Sistema CAD típico.

material de vez en cuando. En el resto del capítulo comentaremos los conceptos más básicos necesarios para escribir código sencillo en VHDL.

2.10.1 REPRESENTACIÓN DE SEÑALES DIGITALES EN VHDL

Cuando se usan herramientas CAD para sintetizar un circuito lógico el diseñador puede proporcionar la descripción inicial de varias formas, como se explicó en la sección 2.9.1. Un modo eficiente consiste en escribir esta descripción en código fuente de VHDL. El compilador de VHDL traduce ese código en un circuito lógico. Cada señal lógica del circuito se representa en el código de VHDL como un objeto de datos. Así como las variables declaradas en cualquier lenguaje de programación de alto nivel tienen tipos asociados —enteros o caracteres, por ejemplo—, los objetos de datos en VHDL pueden ser de varios tipos. La norma original de VHDL, la IEEE 1076, incluye un tipo de datos llamado *BIT*. Un objeto de este tipo es adecuado para representar señales digitales, pues sólo puede tener dos valores, 0 y 1. En este capítulo todas las señales de los ejemplos serán del tipo BIT. En la sección 4.12 estudiaremos otros tipos de datos, que se listan en el apéndice A.

2.10.2 CÓMO ESCRIBIR CÓDIGO SENCILLO EN VHDL

Daremos un ejemplo para ilustrar cómo escribir código fuente sencillo en VHDL. Considérese el circuito lógico de la figura 2.30. Si queremos escribir código de VHDL para representarlo, el primer paso es declarar las señales de entrada y salida. Esto se hace por medio de un constructo denominado *entidad* (*entity*). En la figura 2.31 aparece una entidad apropiada para este ejemplo.

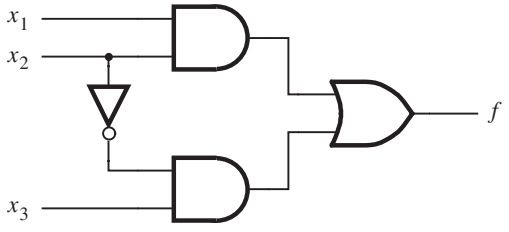


Figura 2.30 Función lógica simple.

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN   BIT ;
           f           : OUT  BIT ) ;
END example1 ;
```

Figura 2.31 Declaración de entidad de VHDL para el circuito de la figura 2.30.

A una entidad debe asignársele un nombre; para este primer ejemplo se eligió el nombre *example1*. Las señales de entrada y salida de la entidad son sus puertos, identificados con la palabra clave PORT. Este nombre se deriva del lenguaje que se ocupa en electrónica en el que la palabra *port* (puerto) se refiere a una conexión de entrada o salida a un circuito electrónico. Cada puerto tiene un *modo* asociado que indica si se trata de una entrada (IN) o de una salida (OUT) del circuito. Cada puerto representa una señal, por tanto tiene un tipo asociado. La entidad *example1* tiene cuatro puertos. Los primeros tres, x_1, x_2 y x_3, son señales de entrada del tipo BIT. El puerto denominado *f* es una salida del tipo BIT.

En la figura 2.31 hemos usado nombres de señal simples, *x*1, *x*2, *x*3 y *f* para los puertos del circuito. Similar a la mayor parte de los lenguajes de programación, VHDL tiene reglas que fijan qué caracteres se permiten en los nombres de señal. Una regla simple es que los nombres de una señal pueden incluir cualquier letra o número, así como el carácter de subrayado '_' (guión bajo). Existen dos requisitos indispensables: un nombre de señal debe comenzar con una letra y no puede ser una palabra clave de VHDL.

Una entidad especifica las señales de entrada y salida para un circuito, pero no proporciona detalle alguno de lo que éste representa. La funcionalidad del circuito debe especificarse con un constructor de VHDL llamado *arquitectura* (*architecture*). En la figura 2.32 aparece una arquitectura para este ejemplo. Debe dársele un nombre; escogimos el de *LogicFunc*. Si bien el nombre puede ser cualquier cadena de texto, es razonable asignar uno significativo para el diseñador. En este caso el nombre elegido fue *LogicFunc* porque la arquitectura especifica la funcionalidad del diseño que usa una expresión lógica. VHDL soporta los operadores booleanos siguientes: AND, OR, NOT, NAND, NOR, XOR y XNOR. (Hasta el momento sólo hemos explicado los operadores AND, OR, NOT, NAND y NOR; el resto se presentará en el capítulo 3.) Después de la palabra clave BEGIN (comienzo), la arquitectura especifica, mediante el operador de VHDL de asignación de señal $\Leftarrow$, que hay que asignar a la salida *f* el resultado de la expresión lógica del miembro derecho del operador. Puesto que VHDL no supone precedencia alguna de los operadores lógicos, se usan paréntesis en la expresión. Cabría esperar que una instrucción de asignación como

$$f <= x1 \text{ AND } x2 \text{ OR NOT } x2 \text{ AND } x3$$

implicaría los paréntesis que siguen

$$f <= (x1 \text{ AND } x2) \text{ OR } ((\text{NOT } x2) \text{ AND } x3)$$

Pero para el código de VHDL tal suposición no es cierta. De hecho, sin los paréntesis el compilador VHDL produciría un error de tiempo de compilación para esta expresión.

La figura 2.33 muestra todo el código de VHDL para este ejemplo, con el que se ilustró el hecho de que un archivo de código fuente de VHDL tiene dos secciones principales: una entidad y una arquitectura.

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figura 2.32 Arquitectura (architecture) de VHDL para la entidad de la figura 2.31.

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN   BIT ;
           f           : OUT  BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figura 2.33 Código completo de VHDL para el circuito de la figura 2.30.

```
ENTITY example2 IS
    PORT ( x1, x2, x3, x4  : IN   BIT ;
           f, g            : OUT  BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
    f <= (x1 AND x3) OR (NOT x3 AND x2) ;
    g <= (NOT x3 OR x1) AND (NOT x3 OR x4) ;
END LogicFunc ;
```

Figura 2.34 Código de VHDL para una función de cuatro entradas.

Una analogía simple para lo que representa cada sección es que la entidad equivale a un símbolo en un diagrama esquemático y la arquitectura especifica los circuitos lógicos en el interior del símbolo.

En la figura 2.34 se presenta un segundo ejemplo de código de VHDL. Este circuito tiene cuatro señales de entrada, $x1$, $x2$, $x3$ y $x4$, y dos señales de salida, f y g. Se asigna a cada salida una expresión lógica. En la figura 2.35 se muestra un circuito lógico producido por el compilador de VHDL para este ejemplo.

Los dos ejemplos precedentes indican que una forma de asignar un valor a una señal en código de VHDL es mediante una expresión lógica. En terminología de VHDL, una expresión lógica se llama *instrucción de asignación simple*. Más adelante veremos que VHDL también soporta otros tipos distintos de instrucciones de asignación y muchas otras funciones útiles para describir circuitos mucho más complejos.

2.10.3 CÓMO *NO* ESCRIBIR CÓDIGO DE VHDL

Cuando está aprendiendo a usar VHDL u otros lenguajes de descripción de hardware, la tendencia del principiante es escribir código semejante al de un programa para computadora, con muchas variables y ciclos. Es difícil determinar qué circuito lógico producirán las herramientas CAD cuando sinteticen tal código. En esta obra se incluyen más de 100 ejemplos de código

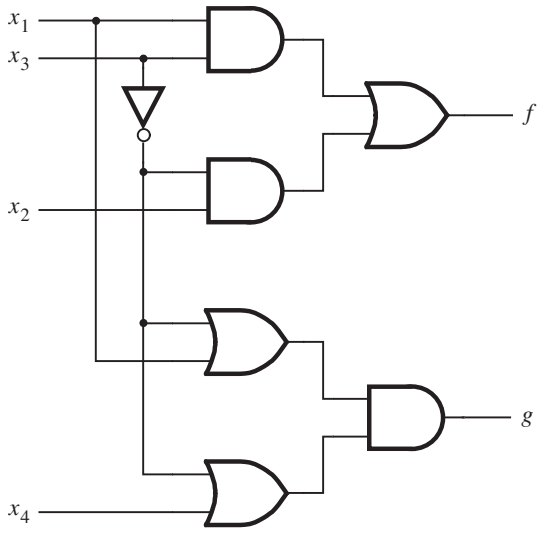


Figura 2.35 Circuito lógico para el código de la figura 2.34.

de VHDL completo que representan numerosos circuitos lógicos de diferentes tipos. En estos ejemplos el código se relaciona fácilmente con el circuito lógico descrito. Se aconseja al lector que adopte el mismo estilo de código. Una buena pauta general es suponer que si el diseñador no puede determinar de inmediato qué circuito lógico describe el código de VHDL, entonces es probable que las herramientas CAD no sinteticen el circuito que se intenta modelar.

Una vez escrito todo el código de VHDL para un diseño específico, lo recomendable es analizar el circuito sintetizado por las herramientas CAD. Se aprende mucho acerca de VHDL, circuitos lógicos y síntesis lógica con este proceso.

2.11 COMENTARIOS FINALES

En este capítulo expusimos el concepto de circuitos lógicos. Demostramos que éstos pueden implementarse mediante compuertas lógicas y que es posible describirlos con un modelo matemático llamado *álgebra booleana*. Puesto que en la práctica los circuitos lógicos suelen ser grandes, es importante contar con buenas herramientas CAD que auxilien al diseñador. El CD que se entrega con el libro incluye el software Quartus II, una herramienta CAD de Altera Corporation. Presentamos algunas características básicas de esta herramienta y alentamos al lector a usar este software en cuanto le fuera posible.

La exposición ha sido muy elemental hasta ahora. En los capítulos siguientes analizaremos tanto los circuitos lógicos como las herramientas CAD de forma pormenorizada. Sin embargo, antes, en el próximo capítulo, examinaremos las tecnologías electrónicas más importantes usadas para construir circuitos lógicos. Con ello brindaremos al lector un cuadro general de las restricciones prácticas que ha de enfrentar un diseñador de circuitos lógicos.

2.12 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas típicos que el lector puede encontrar, al tiempo que se muestra cómo resolverlos.

Ejemplo 2.8

Problema: Determine si la ecuación siguiente es válida

$$\bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2 = \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3$$

Solución: La ecuación es válida si las expresiones de los miembros izquierdo y derecho representan la misma función. A fin de realizar la comparación puede construir una tabla de verdad por cada lado y ver si ambas son iguales. Para hacerlo por medio de álgebra puede derivar una forma de suma canónica de productos por cada expresión.

En virtud de que $x + \bar{x} = 1$ (teorema 8*b*), es posible manipular el miembro izquierdo como sigue:

$$\begin{aligned} \text{LI} &= \bar{x}_1\bar{x}_3 + x_2x_3 + x_1\bar{x}_2 \\ &= \bar{x}_1(x_2 + \bar{x}_2)\bar{x}_3 + (x_1 + \bar{x}_1)x_2x_3 + x_1\bar{x}_2(x_3 + \bar{x}_3) \\ &= \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3 + \bar{x}_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 \end{aligned}$$

Estos términos producto presentan los mintérminos 2, 0, 7, 3, 5 y 4, respectivamente.

Para el miembro derecho se tiene

$$\begin{aligned} \text{LD} &= \bar{x}_1x_2 + x_1x_3 + \bar{x}_2\bar{x}_3 \\ &= \bar{x}_1x_2(x_3 + \bar{x}_3) + x_1(x_2 + \bar{x}_2)x_3 + (x_1 + \bar{x}_1)\bar{x}_2\bar{x}_3 \\ &= \bar{x}_1x_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_3 \end{aligned}$$

Estos términos producto representan los mintérminos 3, 2, 7, 5, 4 y 0, respectivamente. Como ambas expresiones especifican los mismos mintérminos, representan la misma función; por tanto, la ecuación es válida. Otra forma de representar esta función es mediante $\sum m(0, 2, 3, 4, 5, 7)$.

Ejemplo 2.9

Problema: Diseñe la expresión de producto de sumas de costo mínimo para la función $f(x_1, x_2, x_3, x_4) = \sum m(0, 2, 4, 5, 6, 7, 8, 10, 12, 14, 15)$.

Solución: La función se define en términos de sus mintérminos. Para encontrar una expresión POS debemos comenzar con la definición en términos de maxitérminos, que es $f = \prod M(1, 3, 9, 11, 13)$. Por ende,

$$\begin{aligned} f &= M_1 \cdot M_3 \cdot M_9 \cdot M_{11} \cdot M_{13} \\ &= (x_1 + x_2 + x_3 + \bar{x}_4)(x_1 + x_2 + \bar{x}_3 + \bar{x}_4)(\bar{x}_1 + x_2 + x_3 + \bar{x}_4)(\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2 + x_3 + \bar{x}_4) \end{aligned}$$

El producto de los primeros dos maxitérminos puede escribirse de nuevo como

$$\begin{aligned} M_1 \cdot M_3 &= (x_1 + x_2 + \overline{x}_4 + x_3)(x_1 + x_2 + \overline{x}_4 + \overline{x}_3) && \text{por la propiedad conmutativa } 10b \\ &= x_1 + x_2 + \overline{x}_4 + x_3\overline{x}_3 && \text{por la propiedad distributiva } 12b \\ &= x_1 + x_2 + \overline{x}_4 + 0 && \text{por el teorema } 8a \\ &= x_1 + x_2 + \overline{x}_4 && \text{por el teorema } 6b \end{aligned}$$

De manera similar, $M_9 \cdot M_{11} = \overline{x}_1 + x_2 + \overline{x}_4$. Ahora podemos usar de nuevo M_{11}, de acuerdo con la propiedad $7a$, para derivar $M_{11} \cdot M_{13} = \overline{x}_1 + x_3 + \overline{x}_4$. En consecuencia

$$f = (x_1 + x_2 + \overline{x}_4)(\overline{x}_1 + x_2 + \overline{x}_4)(\overline{x}_1 + x_3 + \overline{x}_4)$$

Si aplicamos de nuevo $12b$ se obtiene la respuesta final

$$f = (x_2 + \overline{x}_4)(\overline{x}_1 + x_3 + \overline{x}_4)$$

Ejemplo 2.10

Problema: Un circuito que controla un sistema digital tiene tres entradas: x_1, x_2 y x_3. Debe reconocer tres condiciones:

- La condición A es verdadera si x_3 es verdadera y x_1 es verdadera o x_2 es falsa
- La condición B es verdadera si x_1 es verdadera y x_2 o x_3 son falsas
- La condición C es verdadera si x_2 es verdadera y x_1 es verdadera o x_3 falsa

El circuito de control debe producir una salida de 1 si al menos dos de las condiciones *A, B* y *C* son verdaderas. Diseñe el circuito más simple que pueda usarse para este propósito.

Solución: Sea 1 para verdadero y 0 para falso; entonces las tres condiciones pueden expresarse del modo siguiente:

$$\begin{aligned} A = x_3(x_1 + \overline{x}_2) &= x_3x_1 + x_3\overline{x}_2 \\ B = x_1(\overline{x}_2 + \overline{x}_3) &= x_1\overline{x}_2 + x_1\overline{x}_3 \\ C = x_2(x_1 + \overline{x}_3) &= x_2x_1 + x_2\overline{x}_3 \end{aligned}$$

Luego, la salida deseada del circuito puede expresarse como $f = AB + AC + BC$. Estos términos producto se pueden determinar como:

$$\begin{aligned} AB &= (x_3x_1 + x_3\overline{x}_2)(x_1\overline{x}_2 + x_1\overline{x}_3) \\ &= x_3x_1x_1\overline{x}_2 + x_3x_1x_1\overline{x}_3 + x_3\overline{x}_2x_1\overline{x}_2 + x_3\overline{x}_2x_1\overline{x}_3 \\ &= x_3x_1\overline{x}_2 + 0 + x_3\overline{x}_2x_1 + 0 \\ &= x_1\overline{x}_2x_3 \end{aligned}$$

$$\begin{aligned} AC &= (x_3x_1 + x_3\overline{x}_2)(x_2x_1 + x_2\overline{x}_3) \\ &= x_3x_1x_2x_1 + x_3x_1x_2\overline{x}_3 + x_3\overline{x}_2x_2x_1 + x_3\overline{x}_2x_2\overline{x}_3 \\ &= x_3x_1x_2 + 0 + 0 + 0 \\ &= x_1x_2x_3 \end{aligned}$$

$$\begin{aligned} BC &= (x_1\overline{x}_2 + x_1\overline{x}_3)(x_2x_1 + x_2\overline{x}_3) \\ &= x_1\overline{x}_2x_2x_1 + x_1\overline{x}_2x_2\overline{x}_3 + x_1\overline{x}_3x_2x_1 + x_1\overline{x}_3x_2\overline{x}_3 \\ &= 0 + 0 + x_1\overline{x}_3x_2 + x_1\overline{x}_3x_2 \\ &= x_1x_2\overline{x}_3 \end{aligned}$$

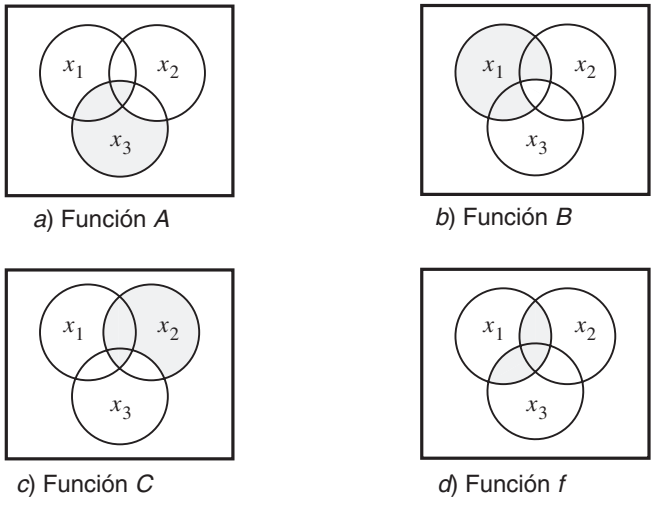
En consecuencia, *f* puede escribirse como

$$\begin{aligned} f &= x_1\overline{x}_2x_3 + x_1x_2x_3 + x_1x_2\overline{x}_3 \\ &= x_1(\overline{x}_2 + x_2)x_3 + x_1x_2(x_3 + \overline{x}_3) \\ &= x_1x_3 + x_1x_2 \\ &= x_1(x_3 + x_2) \end{aligned}$$

Ejemplo 2.11 **Problema:** Resolver el problema del ejemplo 2.10 por medio de diagramas de Venn.

Solución: Los diagramas de Venn para las funciones *A*, *B* y *C* del ejemplo 2.10 se muestran en los incisos *a* a *c* de la figura 2.36. Puesto que la función *f* ha de ser verdadera cuando dos o más de *A*, *B* y *C* sean verdaderas, entonces el diagrama de Venn para *f* se forma identificando las áreas sombreadas comunes en los diagramas de Venn de *A*, *B* y *C*. Cualquiera área sombreada en dos o más de estos diagramas también está sombreada en *f*, como se muestra en la figura 2.36*d*. Este diagrama corresponde a la función

$$f = x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$$



a) Función *A* *b*) Función *B*

c) Función *C* *d*) Función *f*

Figura 2.36 Diagramas de Venn para el ejemplo 2.11.

Ejemplo 2.12

Problema: Derivar la expresión de suma de productos más simple para la función

$$f = x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1\overline{x}_2x_4$$

Solución: La aplicación de la propiedad de consenso 17*a* a los primeros dos términos produce

$$\begin{aligned} f &= x_2\overline{x}_3x_4 + x_1x_3x_4 + x_2x_4x_1x_4 + x_1\overline{x}_2x_4 \\ &= x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1x_2x_4 + x_1\overline{x}_2x_4 \end{aligned}$$

Ahora, al aplicar la propiedad de combinación 14*a* en los últimos dos términos se obtiene

$$f = x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1x_4$$

Finalmente, con la propiedad de absorción 13*a* se llega a

$$f = x_2\overline{x}_3x_4 + x_1x_4$$

Ejemplo 2.13

Problema: Derivar la expresión de producto de sumas más simple para la función

$$f = (\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + x_3 + x_4)$$

Solución: La aplicación de la propiedad de consenso 17*b* a los dos primeros términos conduce a

$$\begin{aligned} f &= (\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + x_3 + \overline{x}_1 + \overline{x}_4)(\overline{x}_1 + x_3 + x_4) \\ &= (\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + x_3 + \overline{x}_4)(\overline{x}_1 + x_3 + x_4) \end{aligned}$$

Ahora, al aplicar la propiedad de combinación 14*b* a los dos últimos términos se obtiene

$$f = (\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + x_3)$$

Finalmente, al aplicar la propiedad de absorción 13*b* al primero y al último término resulta

$$f = (\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + x_3)$$

PROBLEMAS

Al final del libro se encuentran las respuestas a los problemas marcados con un asterisco.

2.1 Use manipulación algebraica para comprobar que $x + yz = (x + y) \cdot (x + z)$. Observe que ésta es la propiedad distributiva, como afirma la identidad 12*b* de la sección 2.5.

2.2 Use manipulación algebraica para comprobar que $(x + y) \cdot (x + \overline{y}) = x$.

2.3 Use manipulación algebraica para comprobar que $xy + yz + \overline{x}z = xy + \overline{x}z$. Observe que ésta es la propiedad de consenso 17*a* de la sección 2.5.

2.4 Use diagramas de Venn para comprobar la identidad del problema 1.

2.5 Use diagramas de Venn para comprobar el teorema de DeMorgan, según se da en las expresiones 15*a* y 15*b* de la sección 2.5.

2.6 Use un diagrama de Venn para comprobar que

$$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3) = x_1 + x_2$$

***2.7** Determine si las expresiones siguientes son válidas, es decir, si los miembros izquierdo y derecho representan la misma función.

a) $\overline{x}_1x_3 + x_1x_2\overline{x}_3 + \overline{x}_1x_2 + x_1\overline{x}_2 = \overline{x}_2x_3 + x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_1x_2x_3$
b) $x_1\overline{x}_3 + x_2x_3 + \overline{x}_2\overline{x}_3 = (x_1 + \overline{x}_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)$
c) $(x_1 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\overline{x}_1 + \overline{x}_3)$

2.8 Trace un diagrama de tiempo para el circuito de la figura 2.19*a*. Muestre las formas de onda que pueden observarse en todos los cables del circuito.

2.9 Repita el problema 2.8 para el circuito de la figura 2.19*b*.

2.10 Use manipulación algebraica para demostrar que para las tres variables de entrada x_1, x_2 y x_3,

$$\sum m(1, 2, 3, 4, 5, 6, 7) = x_1 + x_2 + x_3$$

2.11 Use manipulación algebraica para demostrar que para las tres variables de entrada x_1, x_2 y x_3,

$$\Pi M\,(0, 1, 2, 3, 4, 5, 6) = x_1x_2x_3$$

***2.12** Use manipulación algebraica para hallar la mínima expresión en suma de productos para la función $f = x_1x_3 + x_1\overline{x}_2 + \overline{x}_1x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3$.

2.13 Use manipulación algebraica para encontrar la mínima expresión en suma de productos para la función $f = x_1\overline{x}_2\overline{x}_3 + x_1x_2x_4 + x_1\overline{x}_2x_3\overline{x}_4$.

2.14 Use manipulación algebraica para hallar la mínima expresión de producto de sumas para la función $f = (x_1 + x_3 + x_4) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (x_1 + \overline{x}_2 + \overline{x}_3 + x_4)$.

***2.15** Use manipulación algebraica para encontrar la mínima expresión de producto de sumas para la función $f = (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (\overline{x}_1 + \overline{x}_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3)$.

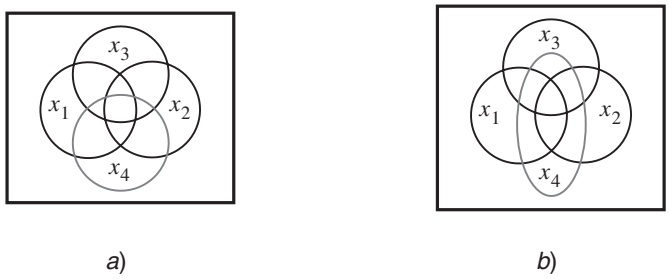
2.16 *a*) Muestre la ubicación de todos los mintérminos en un diagrama de Venn de tres variables.

b) Muestre un diagrama de Venn separado por cada término producto en la función $f = x_1\overline{x}_2x_3 + x_1x_2 + \overline{x}_1x_3$. Use diagramas de Venn para hallar la mínima forma de suma de producto de *f*.

2.17 Represente la función de la figura 2.18 en la forma de diagrama de Venn y determine su mínima forma de suma de productos.

2.18 En la figura P2.1 se muestran dos intentos para trazar un diagrama de Venn para cuatro variables. Para los incisos *a*) y *b*) de la figura, explique por qué el diagrama no es correcto. (Sugerencia: El diagrama de Venn debe ser capaz de representar los 16 mintérminos de las cuatro variables.)

2.19 En la figura P2.2 se observa la representación de un diagrama de Venn de cuatro variables, así como la ubicación de los mintérminos m_0, m_1 y m_2. Muestre la ubicación de los otros mintérminos en el diagrama. Represente la función $f = \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1x_2x_3x_4 + \overline{x}_1x_2$ en este diagrama.



a) *b*)

Figura P2.1 Dos intentos para trazar un diagrama de Venn de cuatro variables.

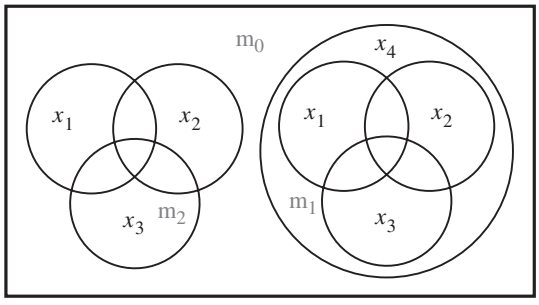


Figura P2.2 Diagrama de Venn de cuatro variables.

***2.20** Diseñe el circuito más simple de suma de productos que implemente la función $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$.

2.21 Diseñe el circuito más simple de suma de productos que implemente la función $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$.

2.22 Diseñe el circuito más simple de producto de sumas que implemente la función $f(x_1, x_2, x_3) = \prod M(0, 2, 5)$.

***2.23** Diseñe el circuito más simple de producto de sumas que implemente la función $f(x_1, x_2, x_3) = \prod M(0, 1, 5, 7)$.

2.24 Derive la expresión más simple de suma de productos para la función $f(x_1, x_2, x_3, x_4) = x_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3$.

2.25 Derive la expresión más simple de suma de productos para la función $f(x_1, x_2, x_3, x_4, x_5) = \bar{x}_1\bar{x}_3\bar{x}_5 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_1x_4x_5 + x_1\bar{x}_2\bar{x}_3x_5$. (Sugerencia: Aplique la propiedad de consenso 17*a*).

2.26 Derive la expresión más simple de producto de sumas para la función $f(x_1, x_2, x_3, x_4) = (\bar{x}_1 + \bar{x}_3 + \bar{x}_4)(\bar{x}_2 + \bar{x}_3 + x_4)(x_1 + \bar{x}_2 + \bar{x}_3)$. (Sugerencia: Aplique la propiedad de consenso 17*b*.)

2.27 Derive la expresión más simple de producto de sumas para la función $f(x_1, x_2, x_3, x_4, x_5) = (\bar{x}_2 + x_3 + x_5)(x_1 + \bar{x}_3 + x_5)(x_1 + x_2 + x_5)(x_1 + \bar{x}_4 + \bar{x}_5)$. (Sugerencia: Aplique la propiedad de consenso 17*b*.)

***2.28** Diseñe el circuito más simple que tenga tres entradas, x_1, x_2 y x_3, que produzca un valor de salida de 1 siempre que dos o más de las variables de entrada tenga el valor 1; de otro modo, la salida debe ser 0.

2.29 Diseñe el circuito más simple que tenga tres entradas, x_1, x_2 y x_3, que produzca un valor de salida de 1 siempre que exactamente una o dos de las variables de entrada tenga el valor de 1; de otro modo, la salida ha de ser 0.

2.30 Diseñe el circuito más simple que tenga cuatro entradas, x_1, x_2, x_3 y x_4, que produzca un valor de salida de 1 siempre que tres o más de las variables de entrada tengan el valor de 1; de otro modo, la salida debe ser 0.

2.31 Para el diagrama de tiempo de la figura P2.3, sintetice la función $f(x_1, x_2, x_3)$ en la forma más simple de suma de productos.

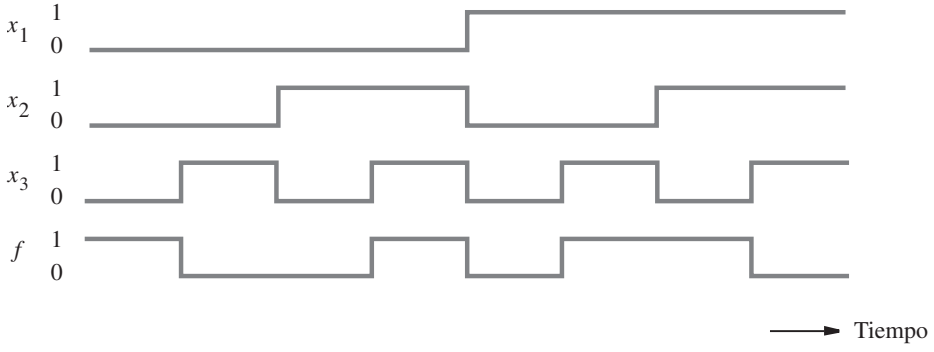


Figura P2.3 Diagrama de tiempo que representa una función lógica.

***2.32** Para el diagrama de tiempo de la figura P2.3, sintetice la función $f(x_1, x_2, x_3)$ en la forma más simple de producto de sumas.

***2.33** Para el diagrama de tiempo de la figura P2.4, sintetice la función $f(x_1, x_2, x_3)$ en la forma más simple de suma de productos.

2.34 Para el diagrama de tiempo de la figura P2.4, sintetice la función $f(x_1, x_2, x_3)$ en la forma más simple de producto de sumas.

2.35 Diseñe un circuito con salida f y entradas x_1, x_0, y_1 y y_0. Sea $X = x_1, x_0$ un número, donde los cuatro valores posibles de X (00, 01, 10 y 11) representan los cuatro números 0, 1, 2 y 3, respectivamente. (La representación de números se explicará en el capítulo 5.) De manera similar, sea $Y = y_1y_0$ la representación de otro número con los mismos cuatro posibles valores. La salida f debe ser 1 si los números representados por X y Y son iguales. De otro modo, f debe ser 0.

a) Elabore la tabla de verdad para *f*.

b) Sintetice la expresión más simple posible de producto de sumas para *f*.

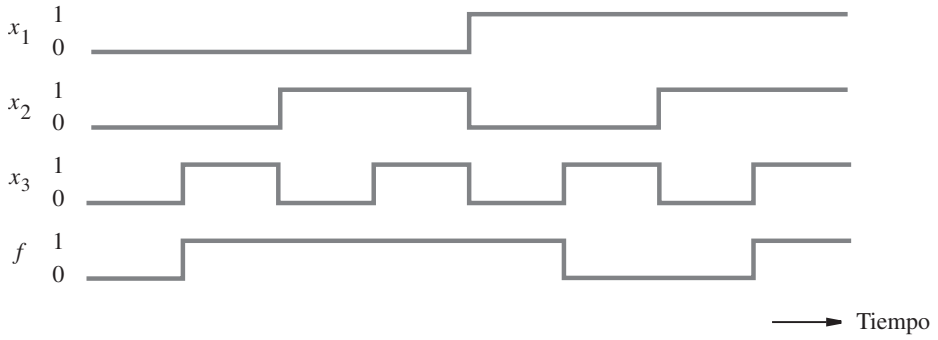


Figura P2.4 Diagrama de tiempo que representa una función lógica.

2.36 Repita el problema 2.35 para el caso en que f debe ser 1 sólo si $X \geq Y$.

a) Elabore la tabla de verdad para f.
b) Muestre la expresión de suma canónica de productos para f.
c) Muestre la expresión más simple posible de suma de productos para f.

2.37 Implemente la función de la figura 2.26 usando sólo compuertas NAND.

2.38 Implemente la función de la figura 2.26 usando solamente compuertas NOR.

2.39 Implemente el circuito de la figura 2.35 usando nada más compuertas NAND y NOR.

***2.40** Diseñe el circuito más simple que implemente la función $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ usando compuertas NAND.

2.41 Diseñe el circuito más simple que implemente la función $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ usando compuertas NAND.

***2.42** Repita el problema 2.40 usando ahora compuertas NOR.

2.43 Repita el problema 2.41 usando ahora compuertas NOR.

2.44 *a*) Use una herramienta de captura esquemática para trazar los esquemas de las funciones siguientes

$$f_1 = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_4 + \overline{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\overline{x}_4 + \overline{x}_1x_2 + x_2x_3$$

b) Use simulación funcional para comprobar que $f_1 = f_2$.

2.45 *a*) Use una herramienta de captura esquemática para trazar los esquemas de las funciones siguientes

$$f_1 = (x_1 + x_2 + \overline{x}_4) \cdot (\overline{x}_2 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$$
$$f_2 = (x_2 + \overline{x}_4) \cdot (x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_4)$$

b) Use simulación funcional para comprobar que $f_1 = f_2$.

2.46 Escriba código de VHDL para implementar la función $f(x_1, x_2, x_3) = \sum m(0, 1, 3, 4, 5, 6)$.

2.47 *a*) Escriba código de VHDL para describir las funciones siguientes

$$f_1 = x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2 + x_1\overline{x}_4$$
$$f_2 = (x_1 + \overline{x}_3) \cdot (x_1 + x_2 + \overline{x}_4) \cdot (x_2 + \overline{x}_3 + \overline{x}_4)$$

b) Use simulación funcional para comprobar que $f_1 = f_2$.

2.48 Considere las instrucciones siguientes de asignación en VHDL

```
f1 <= ((x1 AND x3) OR (NOT x1 AND NOT x3)) OR ((x2 AND x4) OR
      (NOT x2 AND NOT x4)) ;
f2 <= (x1 AND x2 AND NOT x3 AND NOT x4) OR (NOT x1 AND NOT x2 AND x3 AND x4)
      OR (x1 AND NOT x2 AND NOT x3 AND x4) OR
      (NOT x1 AND x2 AND x3 AND NOT x4) ;
```

a) Escriba código de VHDL completo para implementar f1 y f2.

b) Use simulación funcional para comprobar que $f1 = \overline{f2}$.

BIBLIOGRAFÍA

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reimpreso por Dover Publications, Nueva York, 1954.
2. C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", *Transactions of AIEE* 57 (1938), pp. 713-723.
3. E. V. Huntington, "Sets of Independent Postulates for the Algebra of Logic", *Transactions of the American Mathematical Society* 5 (1904), pp. 288-309.
4. S. Brown y Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design* (McGraw-Hill: Nueva York, 2003).
5. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2a. ed. (McGraw-Hill: Nueva York, 1998).
6. D. L. Perry, *VHDL*, 3a. ed. (McGraw-Hill: Nueva York, 1998).
7. J. Bhasker, *A VHDL Primer*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
8. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
9. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).
10. D. J. Smith, *HDL Chip Design* (Doone Publications: Madison, AL, 1996).

capítulo

3

TECNOLOGÍA DE IMPLEMENTACIÓN

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

- Cómo funcionan los transistores y forman interruptores simples
- La tecnología de circuitos integrados
- Las compuertas lógicas CMOS
- Los arreglos de compuertas de campos programables y otros dispositivos lógicos programables
- Las características fundamentales de los circuitos electrónicos

En la sección 1.2 dijimos que los circuitos lógicos se implementan mediante transistores y que existen diversas tecnologías. Ahora estudiaremos éstas con más detalle.

Primero consideremos cómo pueden representarse físicamente las variables lógicas como señales en los circuitos electrónicos. Limitaremos la explicación a variables binarias, que pueden tomar sólo los valores 0 y 1. En un circuito estos valores se representan como niveles de voltaje o de corriente. Ambas posibilidades se usan en diferentes tecnologías. Aquí nos centraremos en la representación más simple y popular: la de niveles de voltaje.

La forma más obvia de representar dos valores lógicos como niveles de voltaje es definir un *voltaje umbral*: el voltaje por abajo del umbral representa un valor lógico y el que está por arriba, otro. La elección acerca de cuál valor lógico se asocia con los niveles de voltaje bajo o alto es arbitraria. Usualmente, el 0 lógico se representa con los niveles de voltaje bajos y el 1 con los altos. Esto se conoce como *sistema de lógica positiva*. La elección opuesta, en la que los niveles de voltaje bajos representan el 1 lógico y los altos el 0 se denomina *sistema de lógica negativa*. En este libro sólo usaremos el sistema de lógica positiva, pero en la sección 3.4 se verá brevemente la negativa.

Cuando se usa el sistema de lógica positiva los valores lógicos 0 y 1 se llaman simplemente "bajo" y "alto". Para aplicar el concepto de voltaje umbral, se definen límites para los niveles bajos y altos, como se muestra en la figura 3.1, que proporciona el voltaje mínimo (V_{SS}) y el máximo (V_{DD}) que puede haber en el circuito. Supóngase que V_{SS} es 0 voltios, lo que corresponde a tierra eléctrica, denotada *Gnd* (del inglés *ground*). El voltaje V_{DD} representa el voltaje de la fuente de poder. Los niveles más comunes para V_{DD} se hallan entre 5 y 1 voltios. En este capítulo emplearemos principalmente el valor V_{DD} = 5 V. En la figura 3.1 se indica que los voltajes en el límite que va de *Gnd* a $V_{0,máx}$ representan el valor lógico 0. El nombre $V_{0,máx}$ significa el nivel de voltaje máximo que un circuito lógico debe reconocer como bajo. De manera similar, el límite que va de $V_{1,mín}$ a V_{DD} corresponde al valor lógico 1, y $V_{1,mín}$ es el nivel de voltaje mínimo que un circuito lógico debe interpretar como alto. Los niveles exactos de

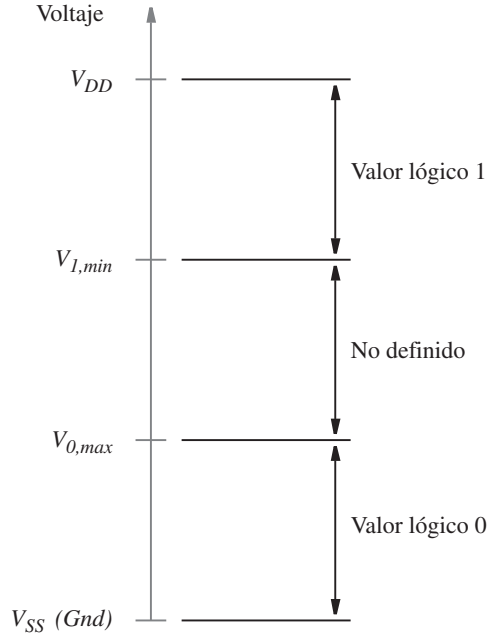


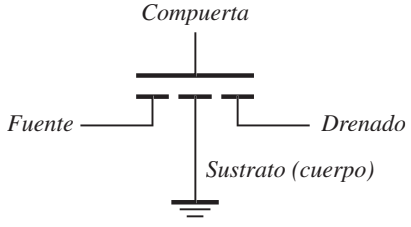
Figura 3.1 Representación de valores lógicos mediante niveles de voltaje.

$V_{0,máx}$ y $V_{1,mín}$ dependen de la tecnología que se use; un ejemplo típico puede establecer $V_{0,máx}$ a 40% de V_{DD} y $V_{1,mín}$ a 60% de V_{DD}. El límite de voltajes entre $V_{0,máx}$ y $V_{1,mín}$ no está definido. Las señales lógicas no suponen voltajes en este límite excepto en la transición de un valor lógico al otro. En la sección 3.8.3 analizaremos con pormenores los niveles de voltaje usados en los circuitos lógicos.

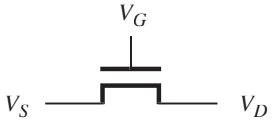
3.1 INTERRUPTORES DE TRANSICIÓN

Los circuitos lógicos se construyen con transistores. En esta obra no se dan detalles del comportamiento de los transistores; ése es un tema de los libros de electrónica, como [1] y [2]. Para entender cómo se construyen los circuitos lógicos podemos suponer que un transistor funciona como un simple interruptor. En la figura 3.2*a* se muestra un interruptor controlado por una señal lógica, x. Cuando x es baja, el interruptor se abre; cuando es alta, el interruptor se cierra. El transistor más popular para implementar un interruptor simple es el *transistor semiconductor de óxido metálico con efecto de campo* (MOSFET, *metal oxide semiconductor field-effect transistor*). Existen dos tipos de MOSFET, conocidos como *canal-n*, que se abrevia *NMOS*, y *canal-p*, que se denota como *PMOS*.

a) Interruptor simple controlado mediante la entrada *x*



b) Transistor NMOS



c) Símbolo simplificado de un transistor NMOS

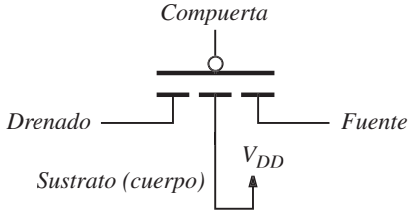
Figura 3.2 Transistor NMOS como interruptor.

En la figura 3.2*b* se presenta el símbolo gráfico de un transistor NMOS. Tiene cuatro terminales eléctricas, llamadas *fuente, drenado, compuerta* y *sustrato*. En los circuitos lógicos la terminal del sustrato (también llamado *cuerpo*) se conecta a *Gnd*. Usaremos el símbolo gráfico simplificado de la figura 3.2*c*, que omite el nodo de sustrato. No hay diferencia física entre las terminales fuente y drenado. En la práctica se distinguen mediante los niveles de voltaje aplicados al transistor; por convención, la terminal con el nivel de voltaje bajo se considera la fuente.

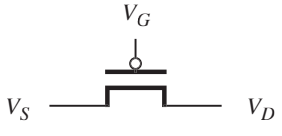
En la sección 3.8.1 brindamos una explicación detallada de cómo funciona el transistor. Por ahora baste saber que está controlado por el voltaje V_G en la terminal de compuerta. Si V_G es bajo, entonces no hay conexión entre la fuente y el drenado, y se dice que el transistor está *apagado*. Si V_G es alto, entonces el transistor está *encendido* y opera como un interruptor cerrado que conecta las terminales fuente y drenado. En la sección 3.8.2 mostraremos cómo calcular la resistencia entre las terminales fuente y drenado cuando el transistor se enciende, pero por el momento supongamos que la resistencia es de 0 Ω.

Los transistores PMOS funcionan de modo opuesto a los NMOS. Estos últimos se usan para realizar el tipo de interruptor que se ilustra en la figura 3.3*a*, donde el interruptor se abre cuando la entrada de control *x* es alta y se cierra cuando es baja. En la figura 3.3*b* se muestra un símbolo. En los circuitos lógicos el sustrato del transistor PMOS siempre está conectado a V_{DD}, lo que

a) Interruptor con el funcionamiento opuesto al de la figura 3.2*a*



b) Transistor PMOS

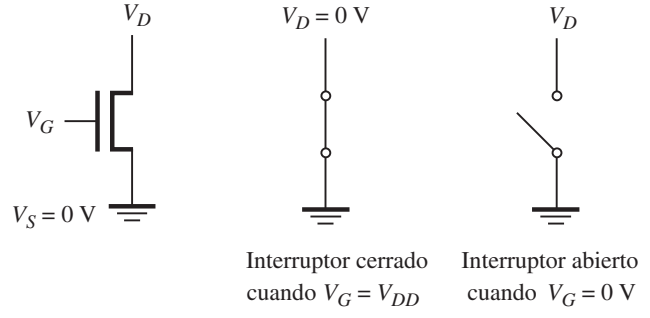


c) Símbolo simplificado de un transistor PMOS

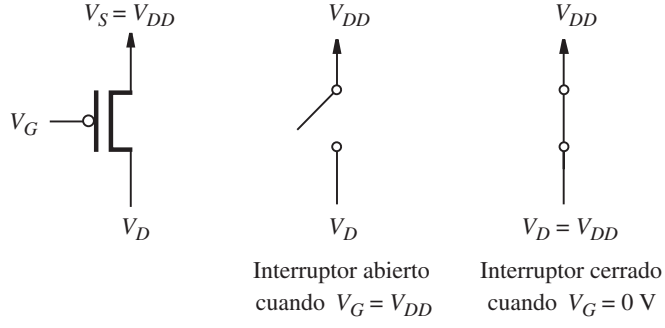
Figura 3.3 Transistor PMOS como interruptor.

conduce al símbolo simplificado de la figura 3.3*c*. Si V_G es alto, entonces el transistor PMOS se apaga y opera como un interruptor abierto. Cuando V_G es bajo, el transistor se enciende y actúa como un interruptor cerrado que conecta la fuente y el drenado. En el transistor PMOS la fuente es el nodo con el voltaje más alto.

En la figura 3.4 se resume el uso típico de los transistores NMOS y PMOS en los circuitos lógicos. Un transistor NMOS se enciende cuando la terminal de compuerta es alta; un transistor PMOS, cuando la compuerta es baja. Cuando el transistor NMOS se enciende, su drenado *baja* hasta *Gnd*; cuando el PMOS se enciende, su drenado *sube* hasta V_{DD}. Debido a cómo funcionan los transistores, un NMOS no sirve para subir su terminal de drenado completamente hasta V_{DD}. De manera similar, un transistor PMOS no sirve para bajar su terminal de drenado completamente hasta *Gnd*. En la sección 3.8 analizaremos con considerable detalle la operación de los MOSFET.



a) Transistor NMOS



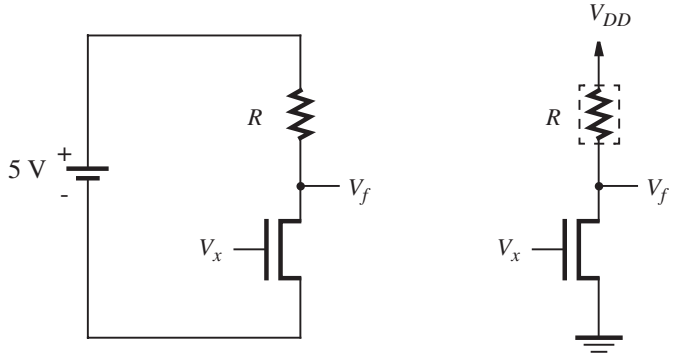
b) Transistor PMOS

Figura 3.4 Transistores NMOS y PMOS en circuitos lógicos.

3.2 COMPUERTAS LÓGICAS NMOS

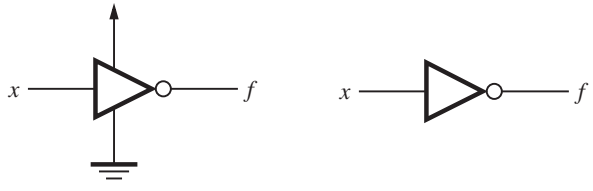
Los primeros esquemas para construir compuertas lógicas con MOSFET se popularizaron en el decenio de 1970; se apoyaban en transistores PMOS o NMOS, mas no en ambos. Desde principios de la década de 1980 se usa una combinación de los dos tipos. Primero describiremos cómo construir circuitos lógicos con transistores NMOS porque es más fácil entender tales circuitos, que se conocen como *circuitos NMOS*. Luego mostraremos cómo se combinan los transistores NMOS y PMOS en la actual tecnología popular conocida como *MOS complementaria* o *CMOS*.

En el circuito de la figura 3.5a, cuando $V_x = 0$ V, el transistor NMOS se apaga. No fluye corriente por el resistor R, y $V_f = 5$ V. Por otra parte, cuando $V_x = 5$ V el transistor se enciende y baja V_f a un nivel de voltaje inferior. En este caso el nivel de voltaje exacto de V_f depende de la cantidad de corriente que fluye por el resistor y el transistor. Casi siempre V_f es aproximadamente 0.2 V (véase la sección 3.8.3). Si V_f se considera una función de V_x, entonces el circuito es una implementación NMOS de una compuerta NOT. En términos lógicos este circuito implementa la función $f = \overline{x}$. En la figura 3.5*b* se presenta el diagrama de circuito simplificado en el que la conexión a la terminal positiva en la fuente de poder se indica mediante una flecha etiquetada con V_{DD}



a) Diagrama de circuito

b) Diagrama de circuito simplificado



c) Símbolos gráficos

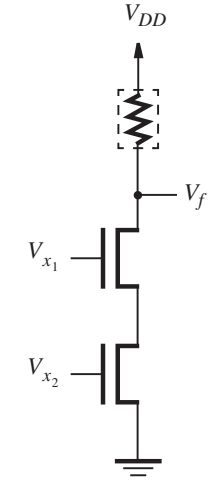
Figura 3.5 Compuerta NOT construida con tecnología NMOS.

y la conexión a la terminal negativa con el símbolo *Gnd*. A lo largo de este capítulo utilizaremos este estilo simplificado de diagrama de circuito.

El propósito del resistor en el circuito de compuerta NOT es limitar la cantidad de corriente que fluye cuando $V_x = 5$ V. En vez de usar un resistor para ello normalmente se emplea un transistor. Abordaremos este tema con cierta profundidad en la sección 3.8.3. En diagramas subsecuentes se trazará un recuadro con líneas discontinuas alrededor del resistor *R* para recordar que se implementa con un transistor.

En la figura 3.5*c* se presentan los símbolos gráficos de una compuerta NOT. El símbolo de la izquierda muestra las terminales de entrada, salida, potencia y tierra, y el de la derecha se simplifica para mostrar únicamente las terminales de entrada y salida. En la práctica sólo se utiliza el símbolo simplificado. Otro nombre que suele emplearse para la compuerta NOT es el de *inversor*. En esta obra usaremos ambos términos.

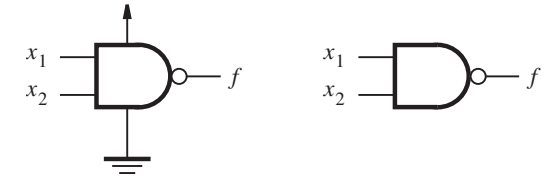
En la sección 2.1 vimos que una conexión de interruptores en serie corresponde a la función lógica AND, en tanto que una conexión en paralelo representa la función OR. Con transistores NMOS es posible implementar la conexión en serie como se describe en la figura 3.6*a*. Si $V_{x_1} = V_{x_2} = 5\ V$, ambos transistores estarán encendidos y V_f estará cerca de 0 *V*. Pero si



a) Circuito

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

b) Tabla de verdad



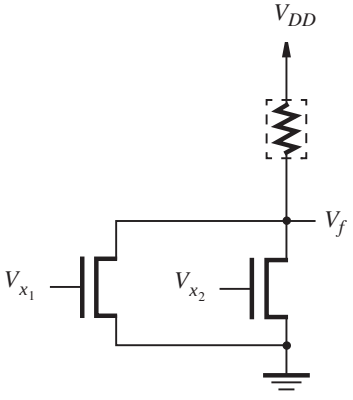
c) Símbolos gráficos

Figura 3.6 Realización NMOS de una compuerta NAND.

V_{x_1} o V_{x_2} es 0, entonces no pasará corriente por los transistores conectados en serie y V_f subirá hasta 5 *V*. La tabla de verdad resultante para *f*, ofrecida en términos de valores lógicos, se muestra en la figura 3.6*b*. La función realizada es el complemento de la función AND, llamada función *NAND* (por NOT-AND). El circuito realiza una compuerta NAND. Sus símbolos gráficos aparecen en la figura 3.6*c*.

En la figura 3.7*a* se presenta la conexión en paralelo de transistores NMOS. Ahí, si $V_{x_1} = 5$ V o $V_{x_2} = 5$ V, entonces V_f estará cerca de 0 V. Sólo si tanto V_{x_1} como V_{x_2} son 0, V_f subirá hasta 5 V. En la figura 3.7*b* se halla la tabla de verdad correspondiente, donde se muestra que el circuito realiza el complemento de la función OR, llamada función NOR (por NOT-OR). Los símbolos gráficos de la compuerta NOR aparecen en la figura 3.7*c*.

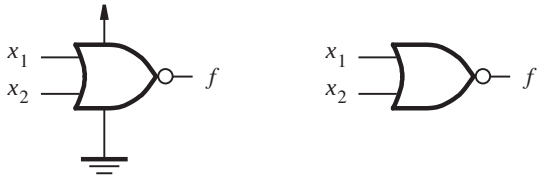
Además de las compuertas NAND y NOR recién descritas, el lector estará naturalmente interesado en las compuertas AND y OR, muy usadas en el capítulo anterior. En la figura 3.8 se indica cómo construir una compuerta AND en tecnología NMOS si se sigue una compuerta NAND con un inversor. El nodo A realiza la NAND de entradas x_1 y x_2, y *f* representa la función AND. De forma similar, una compuerta OR se realiza como una compuerta NOR seguida por un inversor, como se muestra en la figura 3.9.



a) Circuito

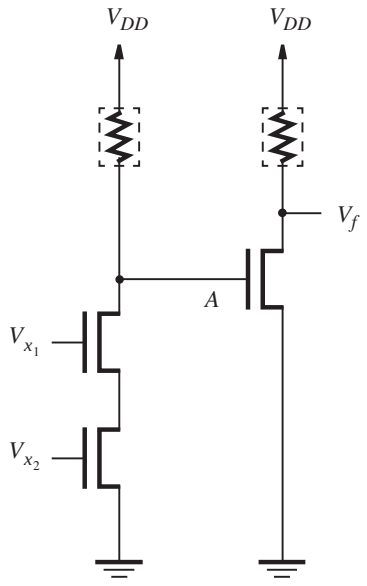
x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

b) Tabla de verdad



c) Símbolos gráficos

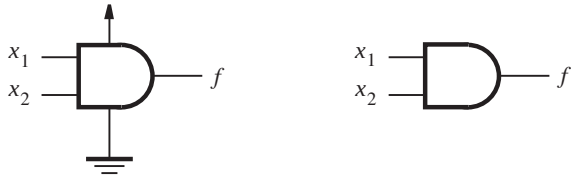
Figura 3.7 Realización NMOS de una compuerta NOR.



a) Circuito

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

b) Tabla de verdad



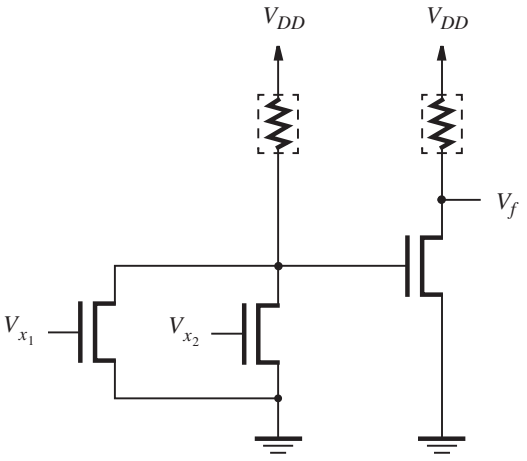
c) Símbolos gráficos

Figura 3.8 Realización NMOS de una compuerta AND.

3.3 COMPUERTAS LÓGICAS CMOS

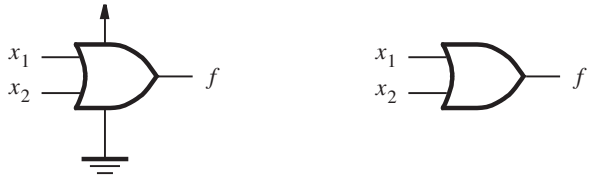
Hasta ahora hemos considerado cómo implementar compuertas lógicas usando transistores NMOS. Para cada uno de los circuitos presentados es posible derivar un circuito equivalente que ocupe transistores PMOS. Sin embargo, es más interesante considerar el uso conjunto de los transistores NMOS y PMOS. El más popular de tal enfoque se conoce como *tecnología CMOS*. En la sección 3.8 veremos que la tecnología CMOS ofrece ciertas ventajas prácticas en comparación con la tecnología NMOS.

En los circuitos NMOS, las funciones lógicas se realizan mediante arreglos de transistores NMOS combinados con un dispositivo de subida que actúa como resistor. Denominaremos *red de bajada* (PDN, *pull-down network*) la parte del circuito que comprende transistores NMOS.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

a) Circuito *b*) Tabla de verdad



c) Símbolos gráficos

Figura 3.9 Realización NMOS de una compuerta OR.

Por tanto, la estructura de los circuitos de las figuras 3.5 a 3.9 pueden caracterizarse por el diagrama de bloques de la figura 3.10. El concepto de circuitos CMOS se basa en la sustitución del dispositivo de subida con una *red de subida* (PUN, *pull-up network*) construida con transistores PMOS, de modo que las funciones realizadas por las redes PDN y PUN se complementan. Luego, un circuito lógico, digamos una compuerta lógica típica, se implementa como se indica en la figura 3.11. Para cualquier valoración de las señales de entrada la PDN baja V_f a *Gnd*, o la PUN sube V_f hasta V_{DD}. Las redes PDN y PUN tienen igual número de transistores, que se arreglan de forma que ambas sean *duales* una de otra. Siempre que la PDN tenga transistores NMOS en serie, la PUN tendrá transistores PMOS en paralelo, y viceversa.

En la figura 3.12 se muestra el ejemplo más simple de circuito CMOS, una compuerta NOT. Cuando $V_x = 0$ V, el transistor T_2 se apaga y el T_1 se enciende. Esto hace que $V_f = 5$ V y como T_2 está apagado, no pasa corriente por los transistores. Cuando $V_x = 5$ V, T_2 está encendido y T_1 apagado. Por ende, $V_f = 0$ V y no fluye corriente porque T_1 está apagado.

Un aspecto clave es que la corriente no fluye en un inversor CMOS cuando la entrada es baja o alta. Esto es cierto para todos los circuitos CMOS; no fluye corriente y, por tanto, no se

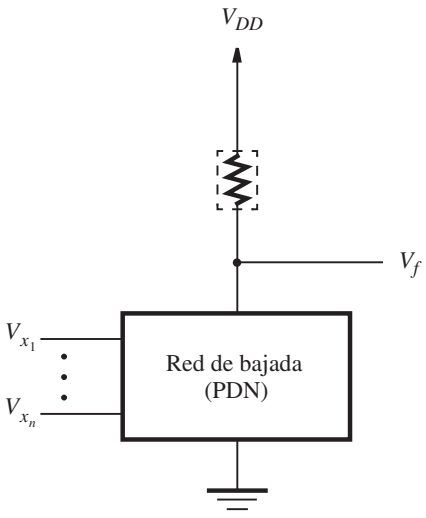


Figura 3.10 Estructura de un circuito NMOS.

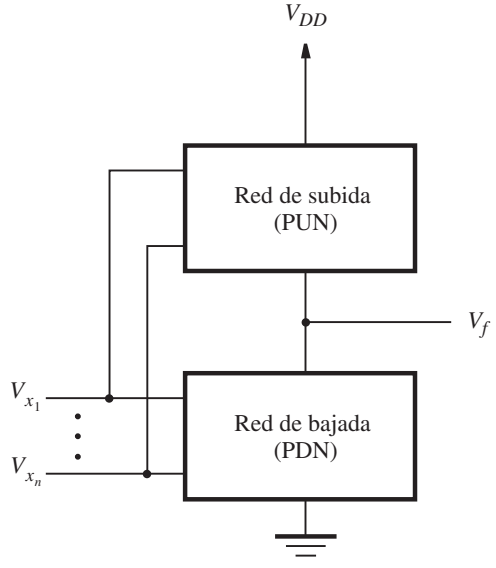
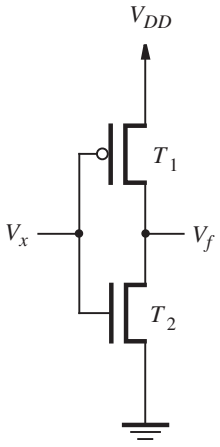


Figura 3.11 Estructura de un circuito CMOS.

disipa potencia en condiciones de estado estacionario. Esta propiedad ocasionó que los CMOS se convirtieran en la tecnología más popular de la actualidad para construir circuitos lógicos. En la sección 3.8 analizaremos con detalle el flujo de corriente y la disipación de potencia.

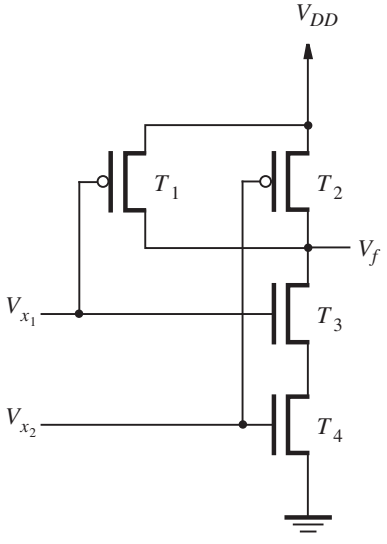
En la figura 3.13 se presenta el diagrama de circuito de una compuerta NAND CMOS. Es similar al circuito NMOS de la figura 3.6, excepto que el dispositivo de subida se sustituyó con la PUN de dos transistores PMOS conectados en paralelo. La tabla de verdad de la figura especifica



x	T_1	T_2	f
0	on	off	1
1	off	on	0

a) Circuito *b*) Tabla de verdad y estados del transistor

Figura 3.12 Realización CMOS de una compuerta NOT.



x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

a) Circuito *b*) Tabla de verdad y estados de transistor

Figura 3.13 Realización CMOS de una compuerta NAND.

el estado de cada uno de los cuatro transistores para cada uno de los valores lógicos de las entradas x_1 y x_2. El lector puede comprobar que el circuito implementa de manera adecuada la función NAND. En condiciones estáticas no existe ruta para el paso de corriente de V_{DD} a *Gnd*.

El circuito de la figura 3.13 puede derivarse de la expresión lógica que define la operación NAND, $f = \overline{x_1 x_2}$. Esta expresión indica las condiciones para las que $f = 1$; por tanto, define la

PUN. Como ésta consta de transistores PMOS, que se encienden cuando sus entradas de control (compuerta) se ponen en 0, una variable de entrada x_i enciende un transistor si $x_i = 0$. Con base en la ley de DeMorgan tenemos que

$$f = \overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$$

Por consiguiente, $f = 1$ cuando cualquier entrada x_1 o x_2 tiene el valor 0, lo que significa que la PUN debe tener dos transistores PMOS conectados en paralelo. La PDN debe implementar el complemento de f, que es

$$\overline{f} = x_1 x_2$$

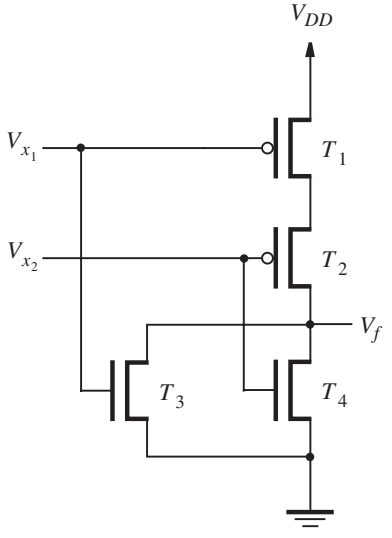
Puesto que $\overline{f} = 1$ cuando tanto x_1 como x_2 son 1, se sigue que la PDN debe tener dos transistores NMOS conectados en serie.

El circuito para una compuerta NOR CMOS se deriva de la expresión lógica que define la operación NOR

$$f = \overline{x_1 + x_2} = \overline{x}_1 \overline{x}_2$$

Como $f = 1$ sólo si x_1 y x_2 tienen el valor 0, entonces la PUN consta de dos transistores PMOS conectados en serie. La PDN, que cumple $\overline{f} = x_1 + x_2$, tiene dos transistores NMOS en paralelo, lo que conduce al circuito mostrado en la figura 3.14.

Una compuerta AND CMOS se construye conectando una compuerta NAND a un inversor, como se ilustra en la figura 3.15. De manera similar, una compuerta OR se construye con una compuerta NOR seguida de una NOT.



a) Circuito

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

b) Tabla de verdad y estados de transistor

Figura 3.14 Realización CMOS de una compuerta NOR.

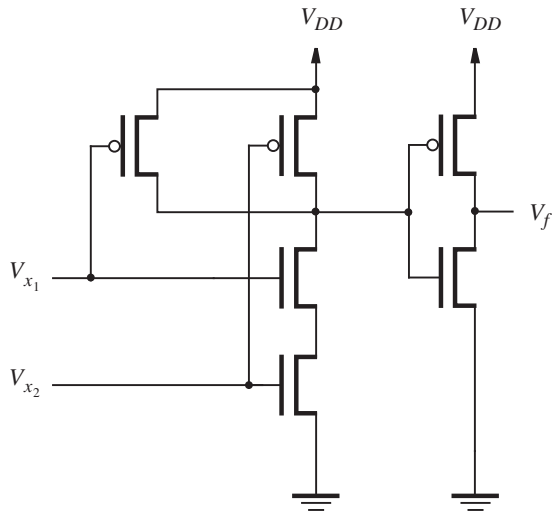


Figura 3.15 Realización CMOS de una compuerta AND.

El procedimiento anterior para derivar un circuito CMOS se aplica a funciones lógicas más generales para crear *compuertas complejas*. Este proceso se ilustra en los dos ejemplos siguientes.

Ejemplo 3.1 Considere la función

$$f = \overline{x}_1 + \overline{x}_2\overline{x}_3$$

Como todas las variables aparecen en su forma de complemento, podemos derivar directamente la PUN. Ésta consta de un transistor PMOS controlado por x_1 en paralelo con una combinación en serie de transistores PMOS controlados por x_2 y x_3. Para la PDN tenemos que

$$\overline{f} = \overline{\overline{x}_1 + \overline{x}_2\overline{x}_3} = x_1(x_2 + x_3)$$

Esta expresión proporciona la PDN que tiene un transistor NMOS controlado por x_1 en serie con una combinación en paralelo de transistores NMOS controlados por x_2 y x_3. El circuito se muestra en la figura 3.16.

Ejemplo 3.2 Considere la función

$$f = \overline{x}_1 + (\overline{x}_2 + \overline{x}_3)\overline{x}_4$$

Entonces

$$\overline{f} = x_1(x_2x_3 + x_4)$$

Estas expresiones conducen directamente al circuito de la figura 3.17.

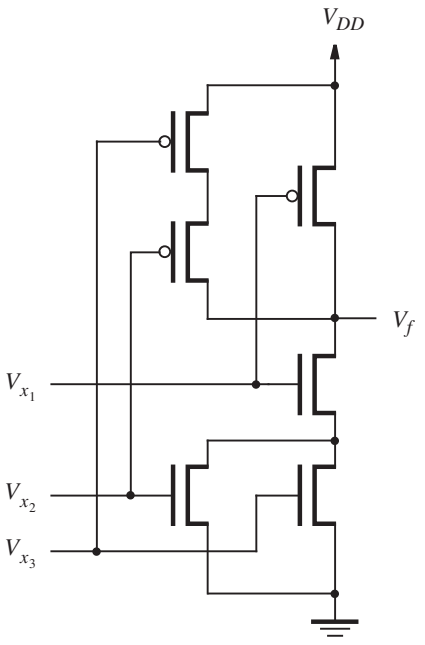


Figura 3.16 Circuito del ejemplo 3.1.

Los circuitos de las figuras 3.16 y 3.17 muestran que es posible implementar funciones lógicas muy complejas combinando conexiones en serie y en paralelo de transistores (que funcionan como interruptores), sin implementar cada conexión en serie o en paralelo como una compuerta AND (mediante la estructura presentada en la figura 3.15) o una OR completas.

3.3.1 VELOCIDAD DE LOS CIRCUITOS DE COMPUERTA LÓGICA

En las secciones precedentes hemos supuesto que los transistores operan como interruptores ideales que no presentan resistencia al flujo de corriente. En consecuencia, aunque derivamos circuitos que cumplen las funciones necesarias en las compuertas lógicas, hemos hecho a un lado el importante tema de la velocidad de operación. En realidad, los interruptores de transistor tienen una resistencia significativa cuando se encienden. Además, los circuitos de transistores incluyen capacitores, que se crean como un efecto colateral del proceso de fabricación. Estos factores influyen en el tiempo requerido para que los valores de señal se propaguen por las compuertas lógicas. En la sección 3.8 ofrecemos una explicación detallada de la velocidad de los circuitos lógicos, así como otros temas prácticos.

3.4 SISTEMA DE LÓGICA NEGATIVA

Al principio del capítulo dijimos que los valores lógicos se representan en dos límites distintos de niveles de voltaje. En el texto usamos la convención de que los niveles de voltaje altos re-

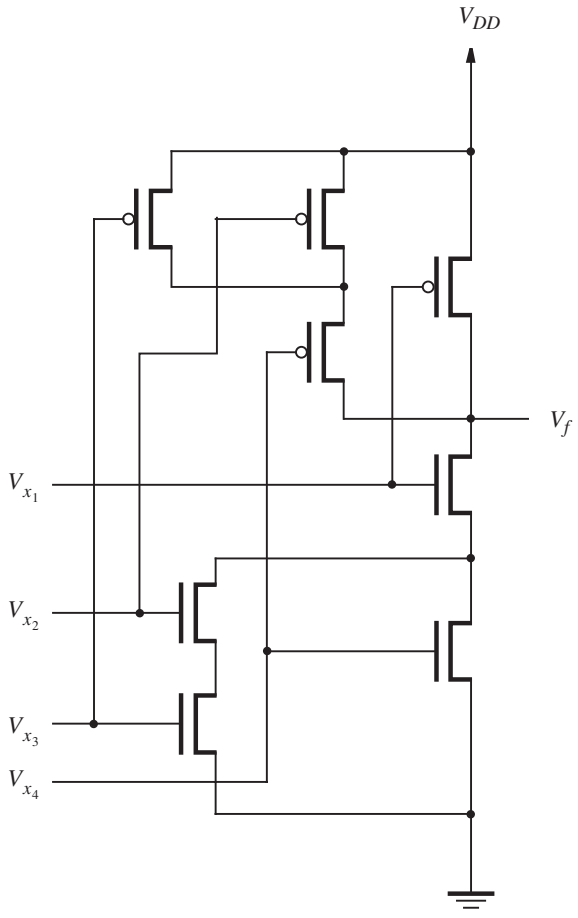
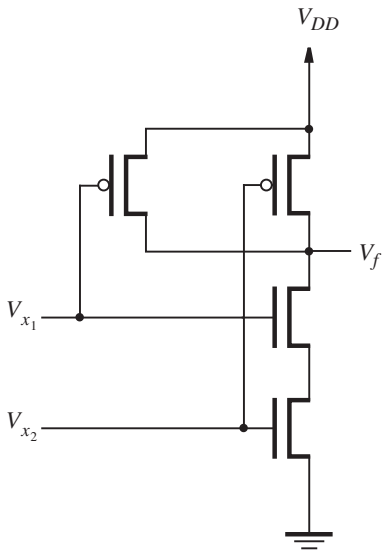


Figura 3.17 Circuito del ejemplo 3.2.

presentan el valor lógico 1 y los bajos, el 0. Esta convención se conoce como *sistema de lógica positiva*, y es el que se utiliza en la mayor parte de las aplicaciones prácticas. En esta sección estudiaremos brevemente el sistema de lógica negativa, en el que se invierte la asociación entre niveles de voltaje y valores lógicos.

Reconsidérese el circuito CMOS de la figura 3.13, que se reproduce en la 3.18*a*. En el inciso *b*) de la figura se da la tabla de verdad del circuito, pero muestra niveles de voltaje en vez de valores lógicos. En esta tabla L se refiere al nivel de voltaje bajo en el circuito, que es 0 V, y H representa el nivel de voltaje alto, que es V_{DD}. Éste es el estilo de la tabla de verdad que los fabricantes de circuitos integrados usan en las hojas de especificaciones para describir la funcionalidad de los chips. Se deja por completo al usuario decidir si L y H se interpretan en términos de valores lógicos como $L = 0$ y $H = 1$, o $L = 1$ y $H = 0$.

En la figura 3.19*a* se presenta la interpretación lógica positiva en la que $L = 0$ y $H = 1$. A partir de lo explicado para la figura 3.13 sabemos que el circuito representa una compuerta NAND de acuerdo con esta interpretación. La interpretación opuesta se muestra en la figura 3.19*b*, donde usamos lógica negativa, de modo que $L = 1$ y $H = 0$. La tabla de verdad indica que



V_{x_1}	V_{x_2}	V_f
L	L	H
L	H	H
H	L	H
H	H	L

a) Circuito *b*) Niveles de voltaje

Figura 3.18 Niveles de voltaje del circuito de la figura 3.13.

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

a) Tabla de verdad en lógica positiva y símbolo de compuerta

x_1	x_2	f
1	1	0
1	0	0
0	1	0
0	0	1

b) Tabla de verdad en lógica negativa y símbolo de compuerta

Figura 3.19 Interpretación del circuito de la figura 3.18.

el circuito representa una compuerta NOR en este caso. Nótese que las filas de la tabla de verdad se citan en el orden opuesto al normalmente usado para ser consistente con los valores *L* y *H* de la figura 3.18*b*. En la figura 3.19*b* también se proporciona el símbolo de compuerta lógica de la compuerta NOR, el cual incluye pequeños triángulos sobre las terminales de ésta para indicar que se empleó el sistema de lógica negativa.

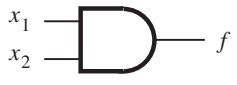
Como otro ejemplo, considérese de nuevo el circuito de la figura 3.15. Su tabla de verdad, en términos de niveles de voltaje, se proporciona en la figura 3.20*a*. Si se utiliza el sistema de lógica positiva este circuito representa una compuerta AND, como se indica en la figura 3.20*b*, pero al usar lógica negativa representa una compuerta OR, como se ilustra en la figura 3.20*c*.

Es posible combinar las lógicas positiva y negativa en un solo circuito, que se conoce como *sistema de lógica mixta*. En la práctica, el sistema de lógica positiva se ocupa en el grueso de las aplicaciones. En esta obra no consideraremos más el sistema de lógica negativa.

V_{x_1}	V_{x_2}	V_f
L	L	L
L	H	L
H	L	L
H	H	H

a) Niveles de voltaje

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1



b) Lógica positiva

x_1	x_2	f
1	1	1
1	0	1
0	1	1
0	0	0

c) Lógica negativa

Figura 3.20 Interpretación del circuito de la figura 3.15.

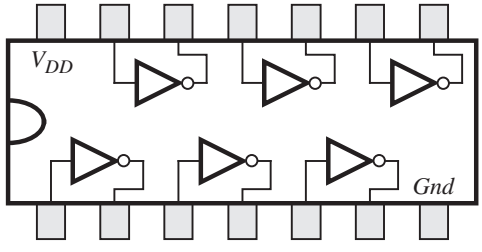
3.5 CHIPS ESTÁNDAR

En el capítulo 1 señalamos que hay muchos tipos de chips de circuitos integrados para implementar circuitos lógicos. Ahora veremos con cierto detalle las opciones disponibles.

3.5.1 CHIPS ESTÁNDAR DE LA SERIE 7400

Un enfoque muy usado hasta mediados de la década de 1980 fue conectar juntos varios chips, cada uno con sólo unas cuantas compuertas lógicas. Hay una amplia variedad de chips, con diferentes tipos de compuertas lógicas, para este propósito. Se conocen como *partes de la serie 7400* porque los números de parte del chip siempre comienzan con los dígitos 74. En la figura 3.21 se da el ejemplo de una parte de la serie 7400. En el inciso *a*) se muestra un tipo de paquete en el que se provee el chip, llamado *paquete en línea doble* (DIP, *dual-inline package*). En el inciso *b*) se ilustra el chip 7404, que comprende seis compuertas NOT. Las conexiones externas del chip se llaman *pines* o *patas*. Para conectar V_{DD} y *Gnd* se usan dos pines, y los otros ofrecen conexiones a las compuertas NOT. Existen muchos chips de la serie 7400, los cuales se describen en los libros de especificaciones editados por los fabricantes respectivos [3-7]. En varios libros de texto también se incluyen los diagramas de algunos de esos chips, como en [8-12].

a) Paquete en línea doble



b) Estructura del chip 7404

Figura 3.21 Un chip de la serie 7400.

Varios fabricantes de circuitos integrados produjeron chips de la serie 7400 en formas estándar, usando especificaciones consensuadas. La competencia entre los diversos fabricantes funciona en ventaja del diseñador, pues tiende a reducir el precio de los chips y a garantizar que siempre hay partes fácilmente disponibles. Por cada chip específico de la serie 7400 se construyen muchas variantes con diferentes tecnologías. Por ejemplo, la parte llamada 74LS00 se construye con una tecnología denominada *lógica transistor-transistor* (TTL, *transistor-transistor logic*), descrita en el apéndice E, mientras que la 74HC00 se fabricó con tecnología CMOS. En general, los chips más populares empleados hoy día son las variantes CMOS.

Como ejemplo del modo en que puede implementarse un circuito lógico con chips de la serie 7400, considérese la función $f = x_1x_2 + \overline{x}_2x_3$, que se muestra en la forma de diagrama lógico en la figura 2.30. Se requiere una compuerta NOT para producir $\overline{x}_2$, así como dos compuertas AND de dos entradas y una compuerta OR también de dos entradas. En la figura 3.22 se muestran tres chips de la serie 7400 que pueden usarse para implementar la función. Supóngase que las tres señales de entrada, x_1, x_2 y x_3 se producen como salidas de algún otro circuito que puede conectarse mediante cables a los tres chips. Nótese que, para los tres chips, se incluyen las conexiones de potencia y tierra. En este ejemplo se utiliza sólo una parte de las compuertas disponibles en los tres chips, por lo que las compuertas restantes sirven para cumplir otras funciones.

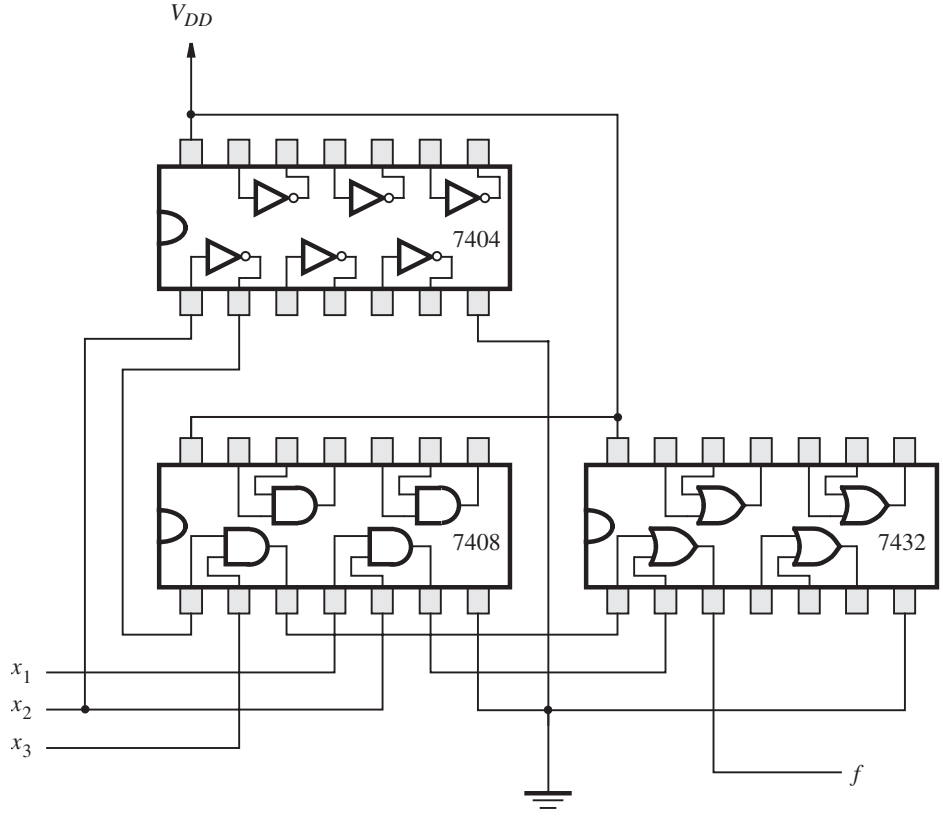


Figura 3.22 Implementación de $f = x_1x_2 + \overline{x}_2x_3$.

Por su baja capacidad lógica, los chips estándar rara vez se usan en la práctica actual, con una excepción. Muchos productos modernos incluyen chips estándar que contienen buffers, que son compuertas lógicas que suelen utilizarse para mejorar la velocidad de los circuitos. Un ejemplo de chip buffer se ilustra en la figura 3.23. Se trata del chip 74244, que comprende ocho *buffer triestado*. En la sección 3.8.8 describimos cómo funcionan los buffers triestado. En vez de indicar cómo se ordenan los buffers en el interior del paquete de chip, como hicimos para las compuertas NOT de la figura 3.21, sólo mostramos los números de pin de los pines del paquete que están conectados a los buffers. El paquete tiene 20 pines, los cuales están numerados de la misma forma que se muestra en la figura 3.21; las conexiones *Gnd* y V_{DD} se proporcionan en los pines 10 y 20, respectivamente. También hay muchos otros chips buffer. Por ejemplo, el chip 162244 tiene 16 buffers triestado. Es parte de una familia de dispositivos similares a los chips de la serie 7400, pero con el doble de compuertas en cada chip. Estos chips están disponibles en múltiples tipos de paquetes; el más popular es el *paquete small-outline integrated circuit* (*SOIC*). Un paquete SOIC tiene forma semejante a la de un DIP, pero su tamaño físico es considerablemente más pequeño.

Conforme la tecnología de los circuitos integrados mejoró, así evolucionó un sistema de clasificación de chips de acuerdo con sus tamaños. Los primeros chips producidos, como los de la serie 7400, comprenden sólo unas cuantas compuertas lógicas. La tecnología empleada para producir dichos chips se conoce como *integración de pequeña escala* (SSI, *small-scale integration*). Los chips que incluyen ligeramente más circuitos lógicos, por lo común alrededor de 10 a 100 compuertas, representan *integración de mediana escala* (*MSI*). Hasta mediados de la década de 1980, los chips muy grandes para considerarse MSI se clasificaron como *integración de gran escala* (*LSI*). En años recientes, el concepto de clasificación de los circuitos según sus tamaños se ha vuelto de poco uso práctico. La mayoría de los circuitos integrados en la actualidad contiene muchos miles o millones de transistores. Sin importar su tamaño exacto, se dice que estos chips grandes están hechos con tecnología de *integración de muy gran escala* (*VLSI*). La tendencia en los productos de hardware digital es integrar cuantos circuitos sea posible en un solo chip. Por ende, la mayor parte de los chips empleados hoy día se construye con tecnología VLSI, y los de tipos de chips más antiguos se usan rara vez.

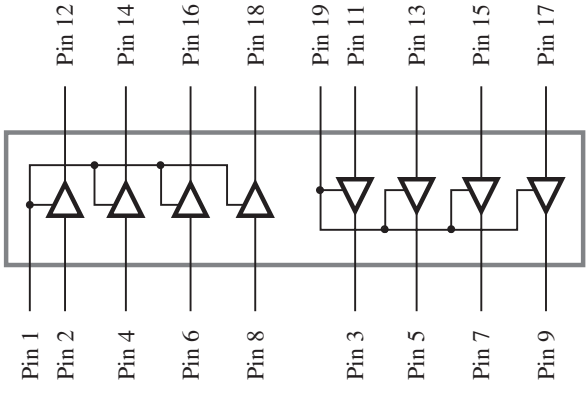


Figura 3.23 Chip buffer 74244.

3.6 DISPOSITIVOS LÓGICOS PROGRAMABLES

La función ofrecida por cada una de las partes de la serie 7400 es fija y no se puede ajustar a una situación de diseño en particular. Este hecho, junto con la limitación de que cada una sólo contiene unas pocas compuertas lógicas, hace que estos chips sean ineficientes para construir circuitos lógicos grandes. Es posible fabricar chips que contengan relativamente grandes cantidades de circuitos lógicos con una estructura que no sea fija. Tales chips se introdujeron por primera vez en el decenio de 1970 y se llaman *dispositivos lógicos programables* (PLD, *programmable logic devices*).

Un PLD es un chip de uso general para implementar circuitos lógicos. Incluye un conjunto de elementos de circuito lógico que pueden adaptarse de diferentes formas. Un PLD puede considerarse una "caja negra" que contiene compuertas lógicas e interruptores programables, como se ilustra en la figura 3.24. Estos últimos permiten que las compuertas lógicas en el interior del PLD se conecten juntas para implementar el circuito lógico que se necesite.

3.6.1 ARREGLO LÓGICO PROGRAMABLE (PLA)

Hay muchos tipos de PLD comerciales. El primero en desarrollarse fue el *arreglo lógico programable* (PLA, *programmable logic array*), cuya estructura general se presenta en la figura 3.25. Con base en la idea de que las funciones lógicas se pueden realizar en forma de suma de productos, un PLA comprende un juego de compuertas AND que alimentan un conjunto de compuertas OR. Como se muestra en la figura, las entradas del PLA, $x_1, \ldots, x_n$ pasan por un grupo de buffers (que proporcionan tanto el valor verdadero como el complemento de cada entrada) hacia un bloque de circuito llamado *plano AND*, o *arreglo AND*. El plano AND produce un juego de términos producto $P_1, \ldots, P_k$, cada uno de los cuales puede configurarse para implementar cualquier función AND de $x_1, \ldots, x_n$. Los términos producto sirven como las entradas a un *plano* OR, que

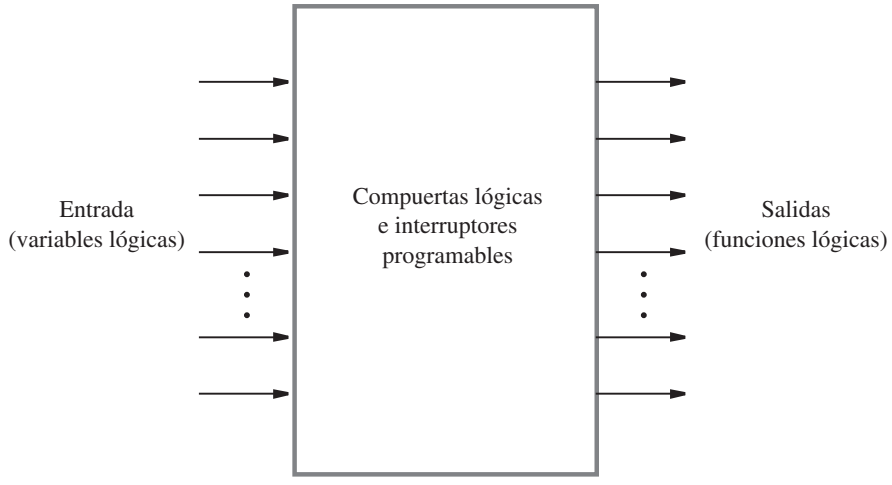


Figura 3.24 Dispositivo lógico programable como caja negra.

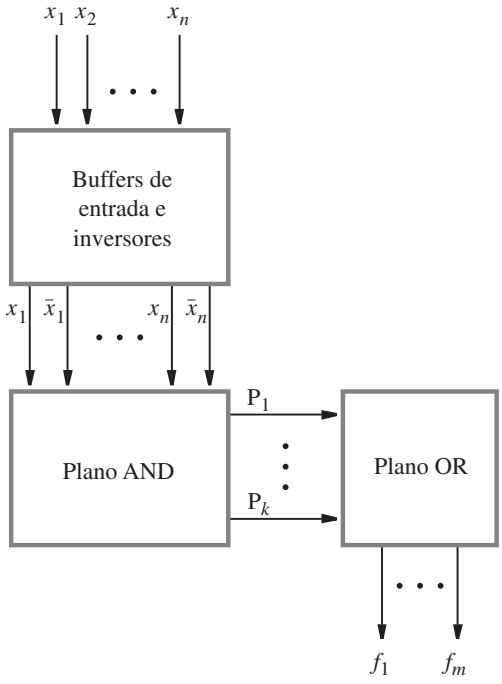


Figura 3.25 Estructura general de un PLA.

produce las salidas $f_1, \ldots, f_m$. Cada salida puede configurarse para realizar cualquier suma de $P_1, \ldots, P_k$ y, por ende, cualquier función de suma de productos de las entradas al PLA.

En la figura 3.26 se encuentra un diagrama más detallado de un pequeño PLA que tiene tres entradas, cuatro términos producto y dos salidas. Cada compuerta AND en el plano AND posee seis entradas, que corresponden a la versiones verdadera y complementada de las tres señales de entrada. Cada conexión a una compuerta AND es programable; la señal que se conecta a una compuerta AND se indica con una línea ondulada, y la que no se conecta, con una línea quebrada. Los circuitos se diseñan de tal modo que cualesquiera entradas no conectadas de la compuerta AND no afecten la salida de la compuerta AND. En los PLA disponibles en el comercio hay varios métodos para hacer las conexiones programables. En la sección 3.10 brindamos una explicación detallada de cómo construir un PLA usando transistores.

En la figura 3.26 la compuerta AND que produce P_1 se muestra conectada a las entradas x_1 y x_2. Por tanto, $P_1 = x_1x_2$. De manera similar, $P_2 = x_1\bar{x}_3$, $P_3 = \bar{x}_1\bar{x}_2x_3$ y $P_4 = x_1x_3$. También hay conexiones programables para el plano OR. La salida f_1 se conecta a los términos producto P_1, P_2 y P_3. Por consiguiente, cumple la función $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$. De igual modo, la salida $f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Aunque la figura 3.26 describe el PLA programado para implementar las funciones descritas líneas arriba, al programar de forma diferente los planos AND y OR, cada una de las salidas f_1 y f_2 podría implementar varias funciones de x_1, x_2 y x_3. La única restricción en cuanto a las funciones que se pueden implementar es el tamaño del plano AND porque éste sólo produce cuatro términos producto. Los PLA disponibles en el comercio vienen en tamaños más grandes de los aquí mostrados. Los parámetros típicos son 16 entradas, 32 términos producto y ocho salidas.

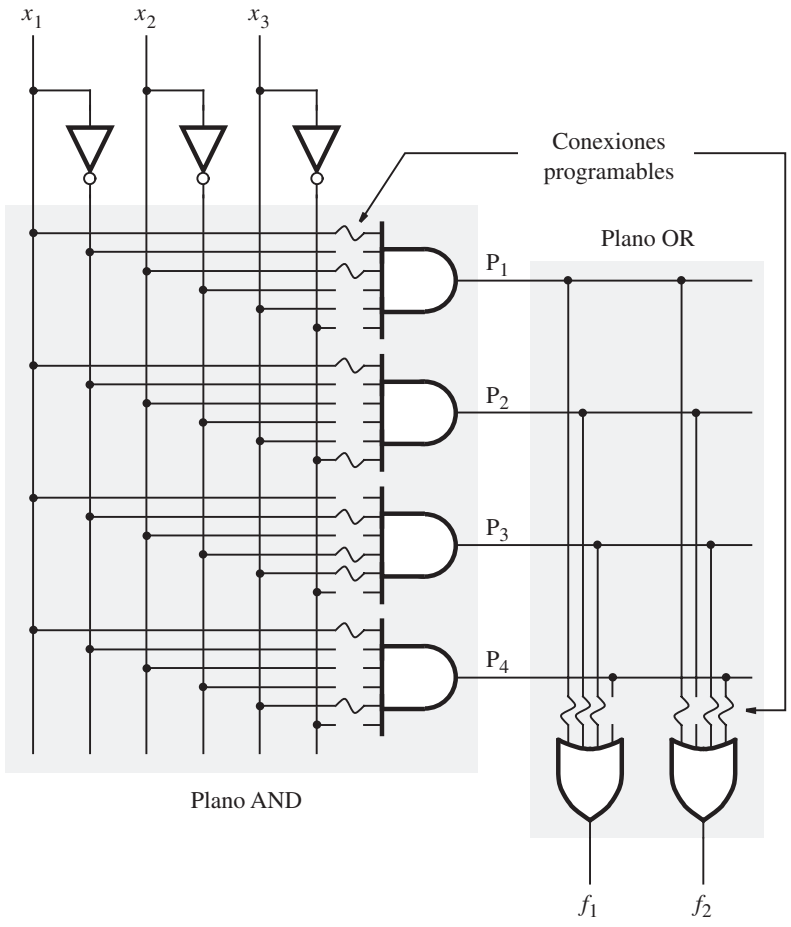


Figura 3.26 Diagrama de nivel de compuerta de un PLA.

Si bien en la figura 3.26 se ilustra con claridad la estructura funcional de un PLA, este estilo de dibujo es complicado para chips más grandes. En su lugar se ha vuelto usual en los libros técnicos usar el estilo de la figura 3.27. Cada compuerta AND se bosqueja como una sola línea horizontal unida a un símbolo de compuerta AND. Las posibles entradas a la compuerta AND se dibujan como líneas verticales que cortan la línea horizontal. En cualquier cruce de una línea vertical y una horizontal puede hacerse una conexión programable, la cual se indica con una X. En la figura 3.27 se muestran las conexiones programables necesarias para implementar los términos producto de la figura 3.26. Cada compuerta OR se traza de forma similar, con una línea vertical unida a un símbolo de compuerta OR. Las salidas de la compuerta AND cortan dichas líneas y se pueden formar las conexiones programables correspondientes. En la figura se advierten las conexiones programables que producen las funciones f_1 y f_2 de la figura 3.26.

El PLA es eficiente en términos del área necesaria para implementarlo en un chip de circuito integrado. Por ello con frecuencia se incluyen PLA como parte de chips más grandes, como los microprocesadores. En este caso se crea un PLA de modo que las conexiones a las compuertas

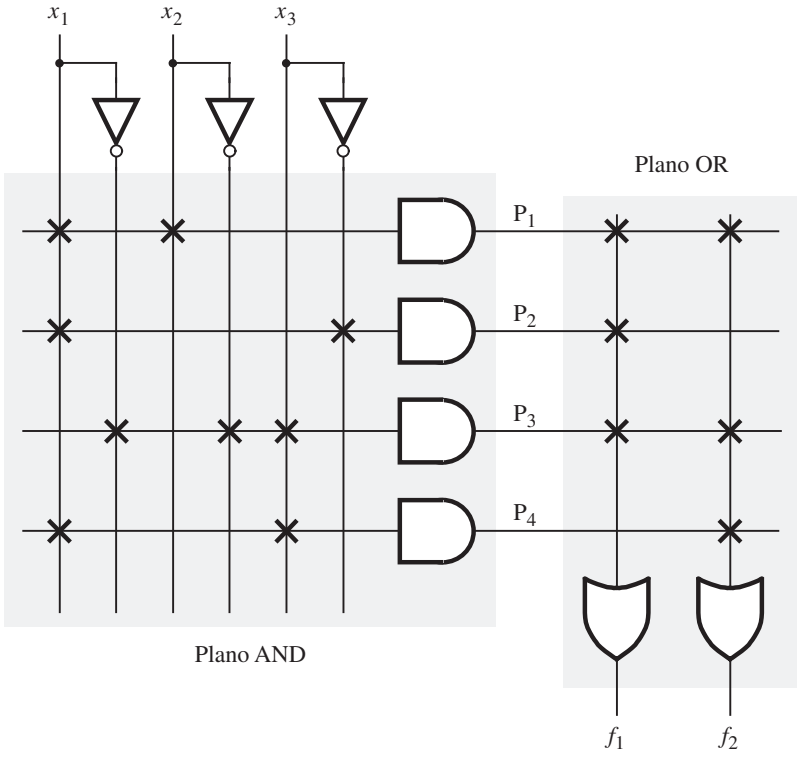


Figura 3.27 Esquema usual para el PLA de la figura 3.26.

AND y OR son fijas, no programables. En la sección 3.10 veremos que las PLA fijas y las programables pueden crearse con estructuras similares.

3.6.2 LÓGICA DE ARREGLO PROGRAMABLE

En un PLA los planos AND y OR son programables. Históricamente, los interruptores programables planteaban dos dificultades a sus fabricantes: eran difíciles de fabricar correctamente y reducían la velocidad de rendimiento de los circuitos implementados en los PLA. Tales desventajas llevaron al desarrollo de un dispositivo similar en el que el plano AND es programable pero el plano OR es fijo. Este chip se conoce como *dispositivo de lógica de arreglo programable* (PAL, *programmable array logic*). Puesto que son más simples de fabricar y por tanto menos costosos que los PLA, aparte de ofrecer mejor rendimiento, las PAL se han vuelto populares en las aplicaciones prácticas.

En la figura 3.28 se presenta un ejemplo de PAL con tres entradas, cuatro términos producto y dos salidas. Los términos producto P_1 y P_2 están cableados a una compuerta OR, y P_3 y P_4 a la otra compuerta OR. La PAL se muestra programada para realizar las dos funciones lógicas $f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$ y $f_2 = \bar{x}_1\bar{x}_2 + x_1x_2x_3$. En comparación con el PLA de la figura 3.27, ésta ofrece menos flexibilidad: el PLA permite hasta cuatro términos producto por compuerta OR,

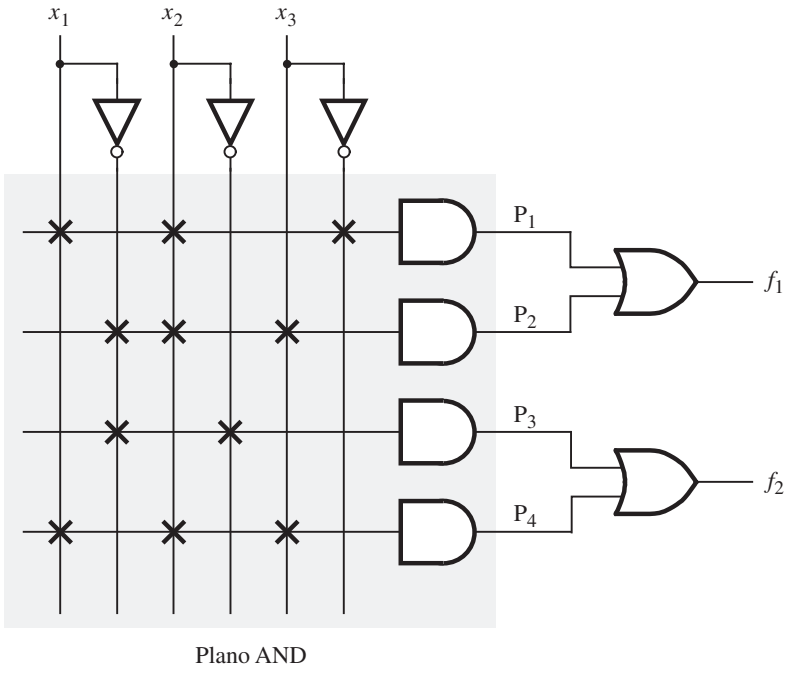


Figura 3.28 Ejemplo de una PAL.

mientras que las compuertas OR de la PAL sólo tienen dos entradas. Para compensar su menor flexibilidad, las PAL se fabrican en diversos tamaños, con varios números de entradas y salidas, y diferentes números de entradas a las compuertas OR. Un ejemplo de PAL comercial se ofrece en el apéndice E.

Hasta ahora hemos supuesto que las compuertas OR en una PAL, como en un PLA, se conectan directamente a los pines de salida del chip. En muchas PAL se agregan circuitos adicionales a la salida de cada compuerta OR para brindar mayor flexibilidad. Es usual emplear el término *macrocelda* para referirse a la compuerta OR combinada con los circuitos adicionales. En la figura 3.29 se muestra un ejemplo de la flexibilidad que puede ofrecerse en una macrocelda. El símbolo etiquetado *flip-flop* representa un elemento de memoria. Almacena el valor producido por la salida de la compuerta OR en un punto particular en el tiempo y puede guardarlo indefinidamente. El flip-flop se controla mediante la señal llamada reloj (*clock*). Cuando el reloj hace una transición del valor lógico 0 al 1, el flip-flop almacena el valor en su entrada *D* en dicho tiempo y este valor aparece en su salida Q. Los flip-flop se usan para implementar muchos tipos de circuitos lógicos, como mostraremos en el capítulo 7.

En la sección 2.8.2 vimos un circuito multiplexor 2 a 1. Tenía dos entradas de datos, una entrada de selección y una salida. La entrada de selección se utiliza para elegir una de las entradas de datos como la salida del multiplexor. En la figura 3.29 un multiplexor 2 a 1 que selecciona como una salida de la PAL la salida de la compuerta OR o la del flip-flop. La línea de selección del multiplexor puede programarse para ser 0 o 1. En la figura 3.29 se muestra otra compuerta lógica, llamada buffer *triestado*, conectada entre el multiplexor y la salida PAL. Estudiaremos

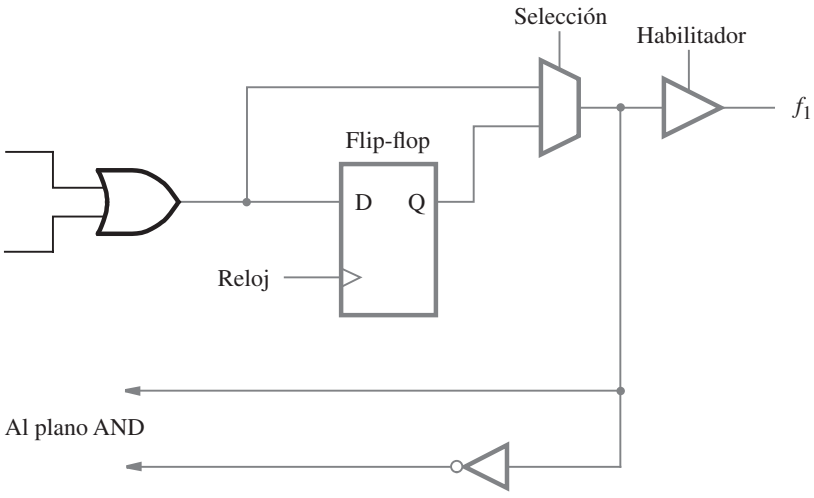


Figura 3.29 Circuitos adicionales agregados a las salidas de la compuerta OR de la figura 3.28.

los buffers triestado en la sección 3.8.8. Finalmente, la salida del multiplexor se "retroalimenta" al plano AND de la PAL. Esta conexión de retroalimentación permite que la función lógica producida por el multiplexor se use internamente en la PAL, lo que a la vez posibilita la implementación de circuitos que tienen múltiples estados, o niveles, de compuertas lógicas.

Varias compañías fabrican PLA o PAL, u otros tipos similares de *PLD simples* (*SPLD*). En el apéndice E se incluye una lista parcial de compañías y los tipos de SPLD que fabrican. El lector interesado puede examinar la información que dichas compañías proporcionan de sus productos y que está disponible en la World Wide Web (WWW). La ubicación WWW de cada compañía se da en la tabla E.1 del apéndice E.

3.6.3 PROGRAMACIÓN DE PLA Y PAL

En las figuras 3.27 y 3.28, cada conexión entre una señal lógica en un PLA o una PAL y las compuertas AND/OR se muestra como una X. En la sección 3.10 describimos cómo implementar estos interruptores con transistores. Los circuitos de usuarios se implementan en los dispositivos mediante la *configuración*, o *programación*, de dichos interruptores. Los chips comerciales contienen unos cuantos miles de interruptores programables; por ende, no es factible que la persona que los usa especifique manualmente el estado de programación deseado de cada interruptor. Para ello se emplean los sistemas CAD. Expusimos las herramientas CAD en el capítulo anterior, donde se describieron métodos para diseñar entradas y la simulación de circuitos. Para los sistemas CAD que soportan el direccionamiento de circuitos a PLD, las herramientas tienen la capacidad de producir automáticamente la información necesaria para programar cada uno de los interruptores en el dispositivo. Un sistema de cómputo que ejecute las herramientas CAD se conecta con un cable a una *unidad de programación* dedicada. Una vez que el usuario termina el diseño de un circuito, las herramientas CAD generan un archivo, llamado *archivo de programación* o *mapa de fusibles*, que especifica el estado que cada interruptor debe tener en el PLD para

realizar correctamente el circuito diseñado. El PLD se coloca en la unidad de programación, y el archivo de programación se transfiere desde el sistema de cómputo. Luego dicha unidad coloca el chip en un *modo de programación* especial y configura interruptor por interruptor. En la figura 3.30 se presenta la fotografía de una unidad de programación. Además de la unidad principal se muestran varios adaptadores; cada uno de ellos se usa para un tipo específico de paquete de chip.

El procedimiento de programación puede tomar unos cuantos minutos para completarse. Usualmente, la unidad de programación puede "volver a leer" de modo automático el estado de cada interruptor, después programarlo para comprobar que el chip se ha programado bien. En los apéndices B, C y D se ofrece una explicación detallada del proceso correspondiente al uso de las herramientas CAD para direccionar los circuitos diseñados a chips programables.

Los PLA o las PAL usados como parte de un circuito lógico por lo general se hallan con otros chips en una tarjeta de circuito impreso (PCB, *printed circuit board*). El procedimiento descrito líneas arriba supone que el chip puede removerse de la tarjeta de circuito para programarlo en la unidad de programación. La remoción se hace posible usando un socket (soporte) en la PCB, como se ilustra en la figura 3.31. Aunque los PLA y las PAL están disponibles en los paquetes DIP presentados en la figura 3.21*a*, también lo están en otro tipo popular de paquete, llamado *plastic-leaded chip carrier* (*PLCC*), que se describe en la figura 3.31. En sus cuatro lados, el paquete PLCC tiene pines que "delimitan" los bordes del chip, en lugar de extenderse rectos hacia abajo como en el caso de un DIP. El socket que alberga el PLCC se une con soldadura a la tarjeta de circuito, y el PLCC se mantiene en el socket mediante fricción.

En vez de depender de una unidad de programación para configurar un chip sería ventajoso poder programarlo mientras aún está unido a su tarjeta de circuito. Este método de programación se llama *programación en el sistema* (ISP, *in-system programming*). No suele ofrecerse para PLA o PAL, pero está disponible para los chips más avanzados que se describen a continuación.

Figura 3.30 Unidad de programación de PLD (cortesía de Data IO Corp.).

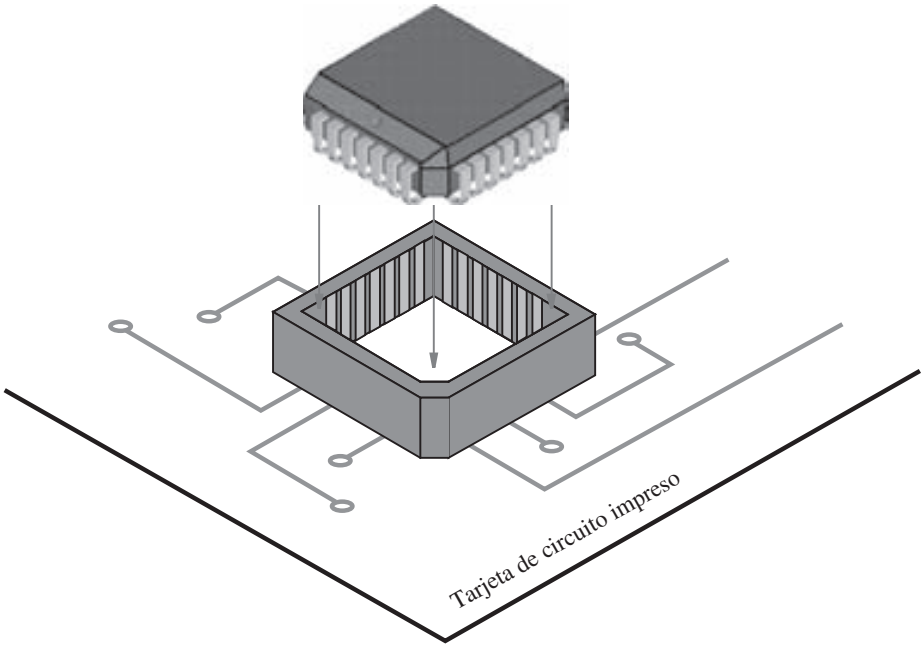


Figura 3.31 Paquete PLCC con socket.

3.6.4 DISPOSITIVOS LÓGICOS PROGRAMABLES COMPLEJOS (CPLD)

Los PLA y las PAL son útiles para implementar una amplia variedad de pequeños circuitos digitales. Cada dispositivo de ésos puede usarse para implementar circuitos que no requieren más que el número de entradas, términos producto y salidas que se ofrecen en el chip específico. Estos chips están limitados a tamaños muy modestos, y soportan una cantidad de entradas más salidas que en combinación no exceden 32. Para la implementación de circuitos que precisan más entradas y salidas se pueden emplear múltiples PLA o PAL o bien un tipo más moderno de chip, llamado *dispositivo lógico programable complejo* (CPLD, *complex programmable logic device*).

Un CPLD comprende múltiples bloques de circuito en un solo chip, con recursos de cableado interno para conectarlos. Cada bloque de circuito es similar a un PLA o a una PAL; nos referiremos a ellos como *bloques parecidos a PAL*. En la figura 3.32 se presenta un ejemplo de CPLD. En ella se incluyen cuatro bloques parecidos a PAL que se conectan a un conjunto de *cables de interconexión*. Cada bloque parecido a PAL también está conectado a un subcircuito etiquetado *bloque de entrada/salida* (*I/O, input/output*) que a su vez está unido a varios de los pines de entrada y salida del chip.

En la figura 3.33 se muestra un ejemplo de la estructura de cableado y las conexiones a un bloque parecido a PAL en un CPLD. El bloque parecido a PAL incluye tres macroceldas (los CPLD reales tienen alrededor de 16 macroceldas en un bloque parecido a PAL), cada una de las cuales consta de una compuerta OR de cuatro entradas (los CPLD reales usualmente proporcionan entre 5 y 20 entradas a cada compuerta OR). La salida de la compuerta OR se conecta a otro tipo de compuerta lógica que aún no hemos explicado, llamada *compuerta OR exclusiva* (XOR), que abordaremos en la sección 3.9.1. El comportamiento de una compuerta XOR es el mismo

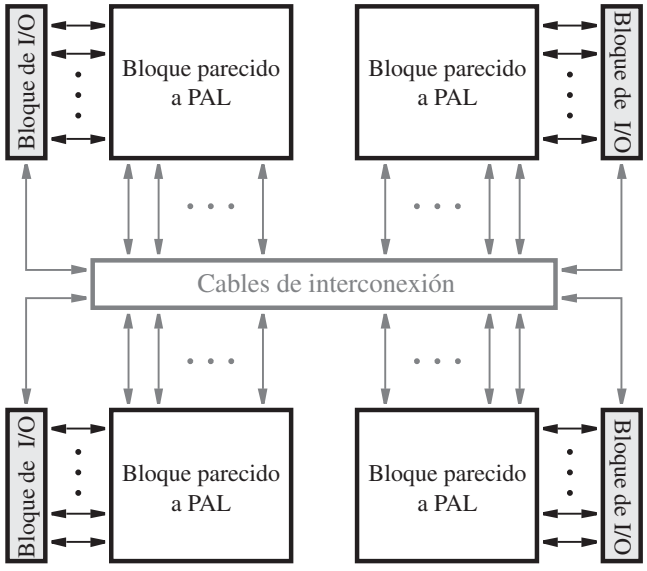


Figura 3.32 Estructura de un dispositivo lógico programable complejo (CPLD).

que el de una compuerta OR, excepto que si ambas entradas son 1, la compuerta XOR produce un 0. Una entrada a la compuerta XOR de la figura 3.33 se puede conectar de manera programable a 1 o a 0; si es 1, entonces la compuerta XOR complementa la salida de la compuerta OR, y si es 0 entonces la compuerta XOR no tiene efecto. La macrocelda también incluye un flip-flop, un multiplexor y un buffer triestado. Como se mencionó en la discusión de la figura 3.29, el flip-flop se usa para almacenar el valor de salida producido por la compuerta OR. Cada buffer triestado (véase sección 3.8.8) se conecta a un pin en el paquete CPLD. El buffer triestado actúa como un interruptor que permite que cada pin se use como una salida del CPLD o como una entrada. Para usar un pin como salida, el correspondiente buffer triestado se habilita, y actúa como interruptor que se enciende. Si el pin se usara como entrada, entonces el buffer triestado se deshabilita, y actúa como un interruptor que se apaga. En este caso una fuente externa puede dirigir una señal hacia el pin, que puede conectarse a otra macrocelda por medio de cable de interconexión.

Los cables de interconexión contienen interruptores programables que se usan para conectar los bloques parecidos a PAL. Cada uno de los cables horizontales puede conectarse a alguno de los verticales que cruza, pero no a todos ellos. Se ha realizado una amplia investigación para decidir cuántos interruptores han de proporcionarse para las conexiones entre los cables. El número de interruptores se elige a fin de ofrecer flexibilidad suficiente para circuitos típicos sin desperdiciar muchos interruptores en la práctica. Cabe advertir que cuando un pin se utiliza como entrada, la macrocelda asociada con ese pin no puede usarse y por tanto se desperdicia. Algunos CPLD incluyen conexiones adicionales entre las macroceldas y el cableado de interconexión que evita desperdiciar macroceldas en tales situaciones.

Los CPLD comerciales varían en tamaño desde sólo dos bloques parecidos a PAL a más de 100. Están disponibles en una diversidad de paquetes, incluido el paquete PLCC que se muestra en la figura 3.31. En la figura 3.34*a* se muestra otro tipo de paquete que se emplea para albergar chips CPLD, llamado *quad flat pack* (QFP). Como un paquete PLCC, el paquete QFP tiene pines

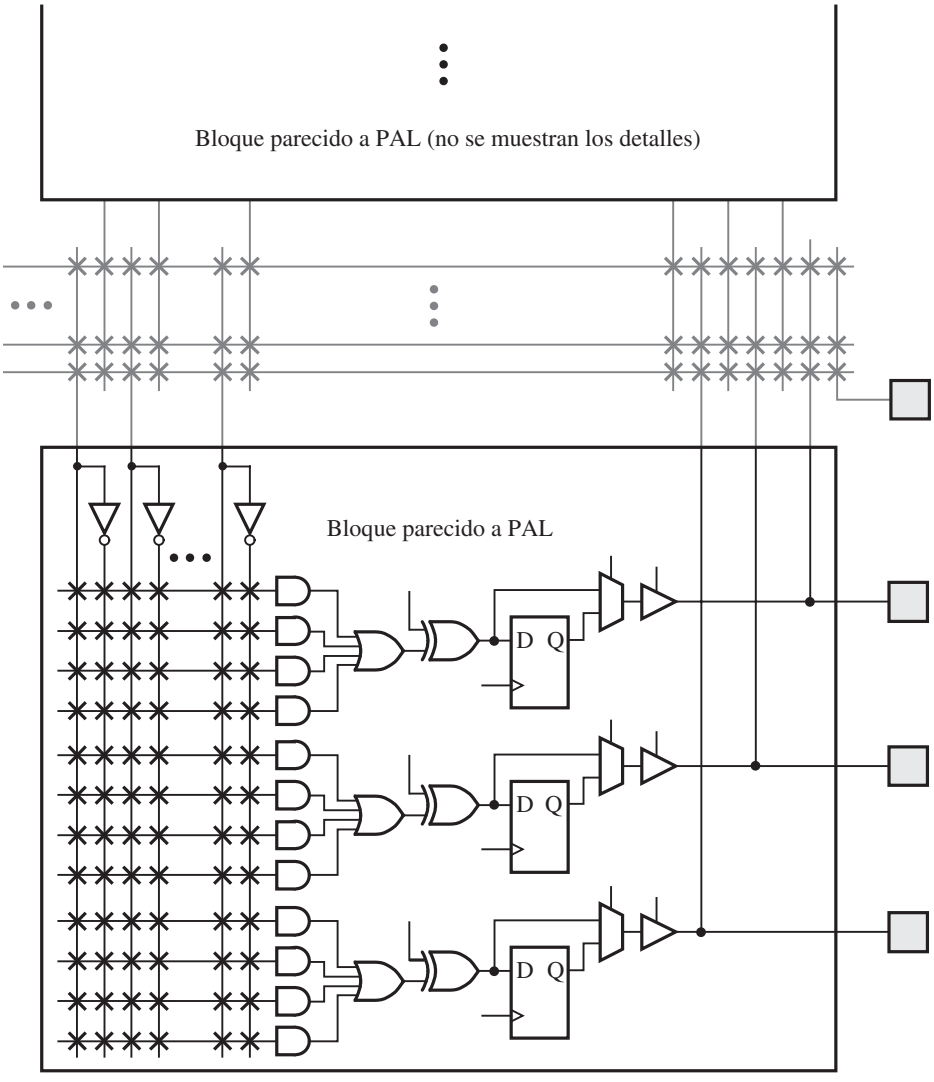
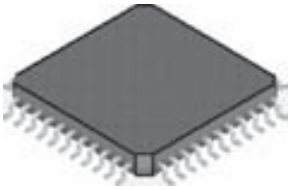


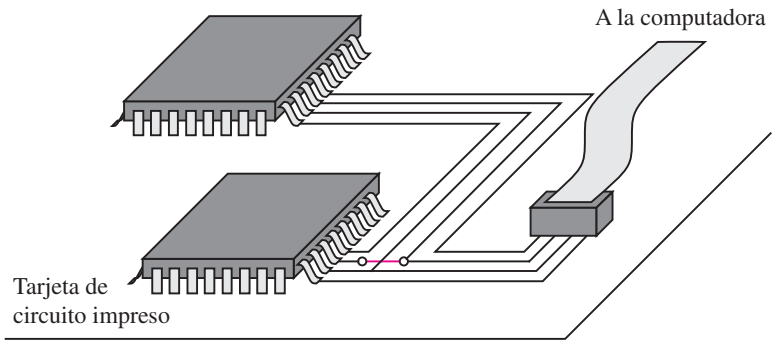
Figura 3.33 Sección del CPLD de la figura 3.32.

en sus cuatro lados, pero mientras que los pines del PLCC delimitan los bordes del paquete, los pines del QFP se extienden hacia afuera desde el paquete, con una forma de curva descendente. Los pines del QFP son mucho más delgados que los de un PLCC, lo que significa que el paquete puede soportar un mayor número de ellos; hay disponibles QFP con más de 200 pines, mientras que los PLCC están limitados a menos de 100 pines.

La mayor parte de los CPLD contiene el mismo tipo de interruptores programables que los usados en los SPLD, que se describen en la sección 3.10. La programación de los interruptores se efectúa con la misma técnica descrita en la sección 3.6.3, en la que el chip se coloca en una unidad de programación de propósito especial. Sin embargo, este método de programación es más bien inconveniente para grandes CPLD por dos razones: primera, los CPLD grandes llegan



a) CPLD en un paquete *quad flat pack* (QFP)



b) Programación JTAG

Figura 3.34 Empaque y programación de CPLD.

a tener más de 200 pines en el paquete del chip, los cuales suelen ser frágiles y se doblan con facilidad; segunda, para programarse en una unidad de programación, se requiere un socket para sostener el chip. Los sockets para grandes paquetes QFP son muy costosos: a veces valen más que el mismo CPLD. Por ello los dispositivos CPLD casi siempre soportan la técnica ISP. En la PCB se incluye un pequeño conector, que alberga el CPLD, y se conecta un cable entre ese conector y una computadora. El CPLD se programa mediante la transferencia por un cable de la información de programación generada por un sistema CAD, de la computadora al CPLD. El IEEE estandarizó los circuitos en el CPLD que permiten este tipo de programación y recibe el nombre de puerto JTAG. Éste usa cuatro cables para transferir información entre la computadora y el dispositivo que se va a programar. El término JTAG *significa Joint Test Action Group* (*grupo de acción de prueba conjunta*). En la figura 3.34*b* se muestra el uso de un puerto JTAG para programar dos CPLD en una tarjeta de circuito. Los CPLD se conectan juntos de modo que ambos pueden programarse con la misma conexión al sistema de cómputo. Una vez que un CPLD se programa, retiene permanentemente el estado programado, aun cuando la fuente de poder del chip se apague. Esta propiedad se llama *programación no volátil*.

Los CPLD se usan para la implementación de muchos tipos de circuitos digitales. En diseños industriales que emplean algún tipo de dispositivo PLD, los CPLD se utilizan mucho, mientras que los SPLD son cada vez menos comunes. Varias compañías ofrecen CPLD compe-

titivos. En la tabla E.2 del apéndice E se listan los nombres de las principales compañías involucradas con sus direcciones web. Se conmina al lector a examinar la información de producto que cada compañía ofrece en sus páginas web. Un ejemplo de un popular CPLD comercial se describe con detalle en el apéndice E.

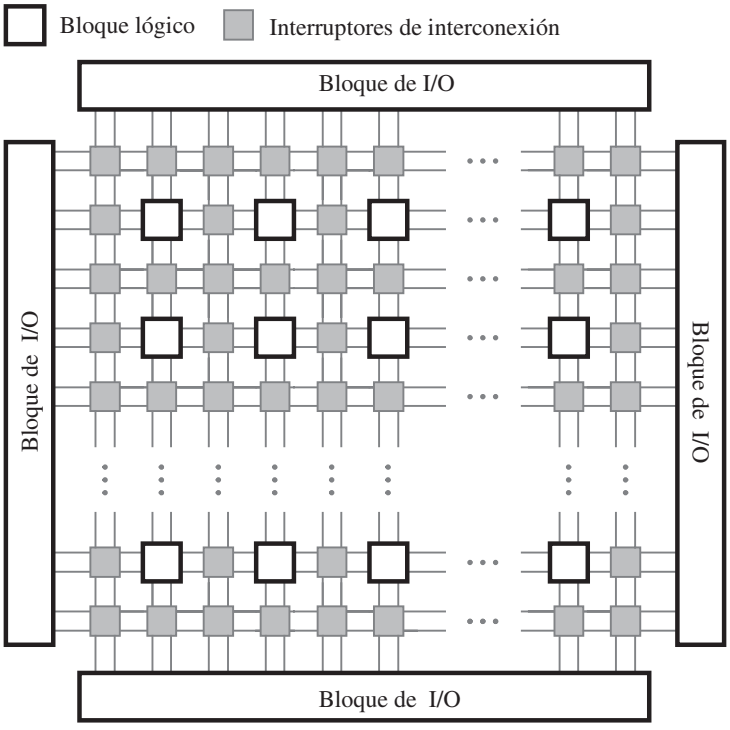
3.6.5 ARREGLOS DE COMPUERTAS DE CAMPOS PROGRAMABLES

Los tipos de chips antes descritos, la serie 7400, los SPLD y los CPLD son útiles para implementar una amplia variedad de circuitos lógicos. Excepto por el CPLD, estos dispositivos son más bien pequeños y adecuados sólo para aplicaciones hasta cierto punto simples. Incluso para los CPLD, nada más pueden acomodarse circuitos lógicos moderadamente grandes en un solo chip. Por razones de costo y rendimiento, es prudente implementar el circuito lógico deseado empleando cuantos menos chips sea posible, de modo que tanto la cantidad de circuitos en un chip como su capacidad funcional son significativos. Una forma de cuantificar el *tamaño* de un circuito es suponer que se construirá usando sólo compuertas lógicas simples y luego estimar cuántas de ellas se necesitan. Una medida que suele usarse es el número total de compuertas NAND de dos entradas que se necesitarían para construir el circuito; esta medida se llama *número de compuertas equivalentes*.

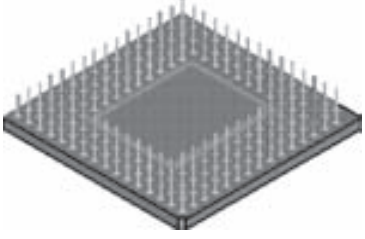
Si utilizamos la métrica de compuertas equivalentes es fácil medir el tamaño de un chip de la serie 7400 porque cada uno contiene únicamente compuertas simples. En el caso de los SPLD y los CPLD la medida típica usada es que cada macrocelda represente aproximadamente 20 compuertas equivalentes. Por ende, una PAL típica que tenga ocho macroceldas puede acomodar un circuito que requiera hasta alrededor de 160 compuertas, y un gran CPLD que tenga 500 macroceldas puede implementar circuitos de hasta más o menos 10 000 compuertas equivalentes.

Según los estándares modernos, un circuito lógico con 10 000 compuertas no es grande. Para implementar circuitos mayores conviene usar un tipo diferente de chip con una capacidad lógica mayor. Un *arreglo de compuertas de campos programables* (FPGA, *field-programmable gate array*) es un dispositivo lógico programable que soporta la implementación de circuitos lógicos hasta cierto punto grandes. Los FPGA son muy diferentes de los SPLD y de los CPLD, ya que no contienen planos AND y OR. En vez de ello, los FPGA ofrecen *bloques lógicos* para la implementación de las funciones requeridas. La estructura general de un FPGA se ilustra en la figura 3.35*a*. Contiene tres tipos principales de recursos: bloques lógicos, bloques de I/O para conectar a los pines del paquete, y cables de interconexión e interruptores. Los bloques lógicos están dispuestos en un arreglo bidimensional, en tanto que los cables de interconexión están organizados como *canales de enrutamiento* horizontales y verticales entre filas y columnas de bloques lógicos. Los canales de enrutamiento contienen cables e interruptores programables que permiten que los bloques lógicos se interconecten de muchas formas. En la figura 3.35*a* se muestran dos ubicaciones de los interruptores programables; los recuadros grises adyacentes a los bloques lógicos sostienen interruptores que conectan las terminales de entrada y salida del bloque lógico a los cables de interconexión, y los recuadros grises que están diagonalmente entre bloques lógicos conectan un cable de interconexión con otro (digamos, un cable vertical con uno horizontal). También hay conexiones programables entre los bloques de I/O y los cables de interconexión. El verdadero número de interruptores programables y cables que hay en un FPGA varía en los chips disponibles en el comercio.

Los FPGA sirven para implementar circuitos lógicos con un tamaño de más de un millón de compuertas equivalentes. Algunos ejemplos de productos FPGA comerciales, de Altera y



a) Estructura general de un FPGA



b) Paquete de arreglo de pines en retícula (PGA) (vista inferior)

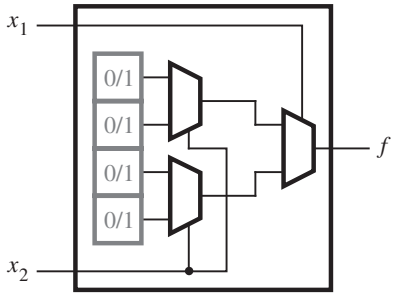
Figura 3.35 Arreglo de compuertas de campos programables (FPGA).

Xilinx, se describen en el apéndice E. Los chips FPGA están disponibles en una variedad de paquetes, incluidos los paquetes PLCC y QFP ya descritos. En la figura 3.35*b* se ilustra otro tipo de paquete, llamado *arreglo de pines en retícula* (PGA, *pin grid array*). Un paquete PGA puede tener hasta unos cuantos cientos de pines en total, que se extienden rectos hacia fuera desde la parte inferior del paquete, en un patrón de retícula. Incluso otra tecnología de empacado que ha surgido se conoce como *ball grid array* (*BGA*). El BGA es similar al PGA excepto que los pines son pequeñas bolas redondas en vez de postes.

La ventaja de los paquetes BGA es que los pines son muy pequeños; por tanto, un paquete hasta cierto punto pequeño puede contener más.

Cada bloque lógico en un FPGA tiene un pequeño número de entradas y salidas. En el mercado hay varios productos FPGA, que presentan diferentes tipos de bloques lógicos. El más usado de éstos es una *tabla de consulta* (LUT, *lookup table*), que contiene *celdas de almacenamiento* que sirven para implementar una pequeña función lógica. Cada celda puede contener un solo valor lógico, 0 o 1. El valor almacenado se produce como la salida de la celda de almacenamiento. Pueden crearse LUT de varios *tamaños*, en los que el tamaño se define mediante el número de entradas. En la figura 3.36*a* se muestra la estructura de una pequeña LUT. Tiene dos entradas, x_1 y x_2, y una salida, f. Es capaz de implementar cualquier función lógica de dos variables. Como una tabla de verdad de dos variables tiene cuatro filas, esta LUT posee cuatro celdas de almacenamiento. Una celda corresponde al valor de salida de cada fila de la tabla de verdad. Las variables de entrada x_1 y x_2 se usan como las entradas de selección de tres multiplexores, los cuales, según la valoración de x_1 y x_2, eligen el contenido de una de las cuatro celdas como la salida de la LUT. En la sección 2.8.2 expusimos los multiplexores; estudiaremos las celdas de almacenamiento en el capítulo 10.

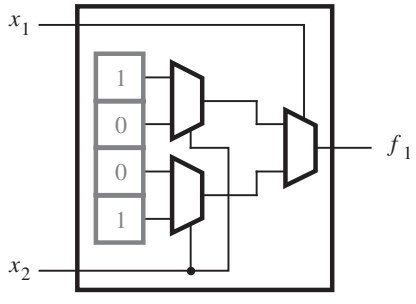
Para ver cómo se realiza una función lógica en la LUT de dos entradas considérese la tabla de verdad de la figura 3.36*b*. La función f_1 de esta tabla puede almacenarse en la LUT como se



a) Circuito para una LUT de dos entradas

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$



c) Contenido de las celdas de almacenamiento en la LUT

Figura 3.36 Tabla de consulta (LUT) de dos entradas.

ilustra en la figura 3.36*c*. El ordenamiento de los multiplexores de la LUT realiza correctamente la función f_1. Cuando $x_1 = x_2 = 0$, la salida de la LUT la dirige la celda de almacenamiento superior, que en la tabla de verdad representa la entrada para $x_1x_2 = 00$. De manera similar, para todos los valores de x_1 y x_2, el valor lógico almacenado en la celda de almacenamiento correspondiente a la entrada en la tabla de verdad elegida por el valor particular aparece en la salida de la LUT. Dar acceso al contenido de las celdas de almacenamiento es sólo una forma en la que los multiplexores pueden usarse para implementar funciones lógicas. En el capítulo 6 explicaremos pormenorizadamente sus aplicaciones.

En la figura 3.37 se presenta una LUT de tres entradas. Tiene ocho celdas de almacenamiento porque una tabla de verdad de tres variables tiene ocho filas. En los chips comerciales de FPGA, las LUT tienen cuatro o cinco entradas, que requieren 16 y 32 celdas de almacenamiento, respectivamente. En la figura 3.29 mostramos que las PAL tienen casi siempre circuitos adicionales incluidos con sus compuertas AND-OR. Lo mismo ocurre con los FPGA, que por lo general tienen circuitos adicionales, además de una LUT, en cada bloque lógico. En la figura 3.38 se muestra cómo incluir un flip-flop en un bloque lógico FPGA. Como explicamos para la

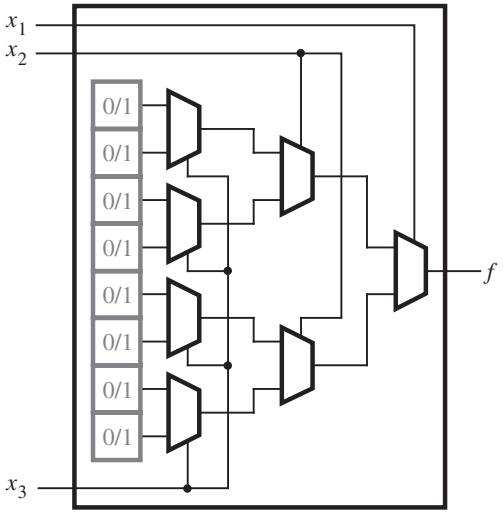


Figura 3.37 Una LUT de tres entradas.

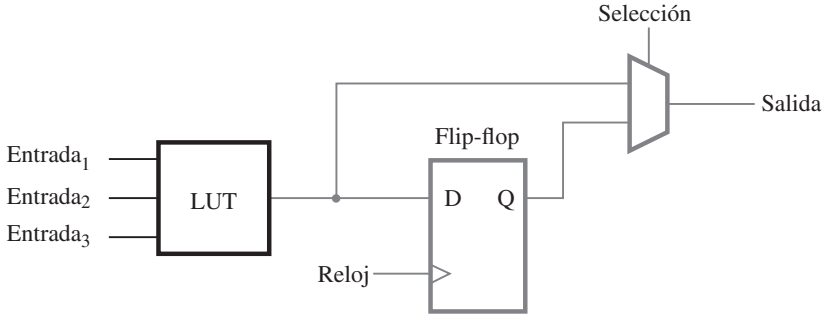


Figura 3.38 Inclusión de un flip-flop en un bloque lógico de un FPGA.

figura 3.29, el flip-flop se usa para almacenar el valor de su entrada *D* bajo el control de su entrada *reloj*. En el apéndice E se presentan ejemplos de bloques lógicos en los FPGA comerciales.

Para realizar un circuito lógico en un FPGA, cada función lógica del circuito ha de ser lo suficientemente pequeña para encajar un solo bloque lógico. En la práctica, el circuito de un usuario se traduce de manera automática a la forma requerida con las herramientas CAD (véase el capítulo 12). Cuando se implementa un circuito en un FPGA, los bloques lógicos se programan para cumplir las funciones necesarias y los canales de enrutamiento para realizar las interconexiones requeridas entre bloques lógicos. Los FPGA se configuran con el método ISP, explicado en la sección 3.6.4. Las celdas de almacenamiento en las LUT de un FPGA son *volátiles*, lo que significa que pierden el contenido que almacenan siempre que la fuente de poder para el chip se apague. Por tanto, el FPGA debe programarse cada vez que se aplique potencia. Con frecuencia, en la tarjeta de circuito que alberga el FPGA se incluye un pequeño chip de memoria que conserva permanentemente sus datos, llamado *memoria programable de sólo lectura* (PROM, *programmable read-only memory*). Las celdas de almacenamiento del FPGA se cargan de modo automático del PROM cuando se aplica potencia a los chips.

En la figura 3.39 se ilustra un pequeño FPGA programado para implementar un circuito. Tiene LUT de dos entradas, y hay cuatro alambres en cada canal de enrutamiento. En la figura se muestran los estados programados tanto de los bloques lógicos como de los interruptores cableados en una sección del FPGA. Los interruptores cableados programables se indican con una X. Cada interruptor mostrado en gris se enciende y realiza una conexión entre un cable horizontal y uno vertical. Los interruptores que se muestran en negro están apagados. En la

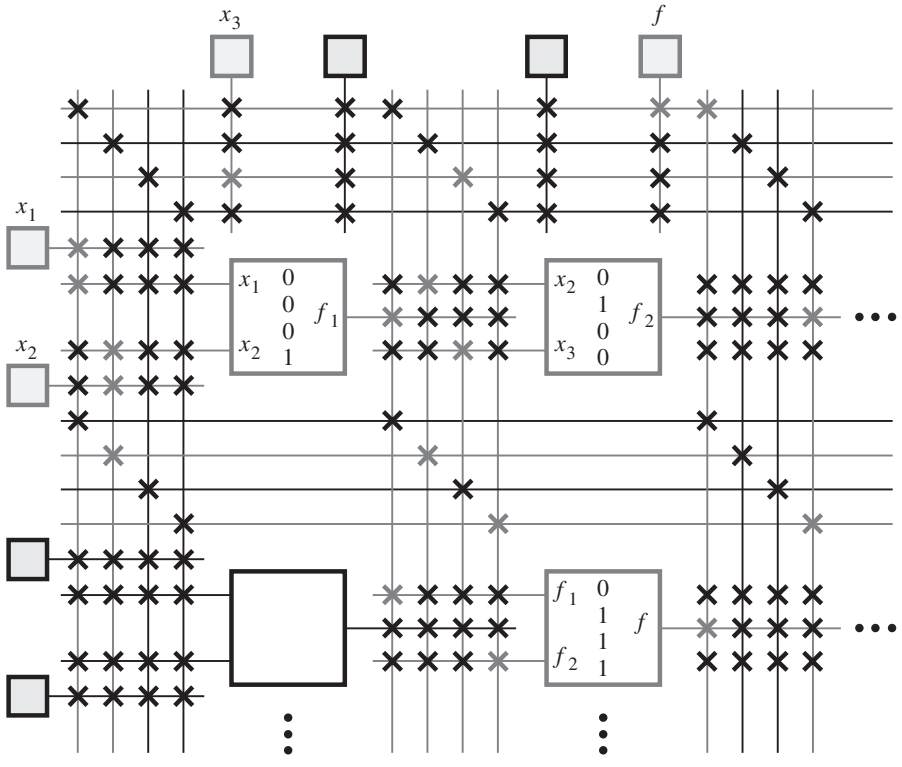


Figura 3.39 Sección de un FPGA programado.

sección 3.10.1 describimos cómo se implementan los interruptores mediante transistores. Las tablas de verdad programadas en los bloques lógicos de la fila superior del FPGA corresponden a las funciones $f_1 = x_1x_2$ y $f_2 = \bar{x}_2x_3$. El bloque lógico de la esquina inferior derecha de la figura se programó para producir $f = f_1 + f_2 = x_1x_2 + \bar{x}_2x_3$.

3.6.6 USO DE HERRAMIENTAS CAD PARA IMPLEMENTAR CIRCUITOS EN CPLD Y FPGA

En la sección 2.9 sugerimos al lector que trabajara con el tutorial 1 del apéndice B para obtener cierta experiencia con herramientas CAD reales. El tutorial 1 cubre los pasos de entrada de diseño y simulación funcional. Ahora que hemos explicado algunos de los detalles de la implementación de circuitos en chips, el lector deseará experimentar aún más con las herramientas CAD. En los tutoriales 2 y 3 (apéndices C y D) mostramos cómo implementar en chips CPLD y FPGA los circuitos diseñados con las herramientas CAD.

3.6.7 APLICACIONES DE LOS CPLD Y FPGA

Los CPLD y FPGA se usan actualmente en muchas aplicaciones diversas, como productos de consumo —reproductores de DVD y aparatos de televisión de alta definición, por ejemplo—, circuitos controladores para fábricas automotrices y equipos de prueba, enrutadores de internet e interruptores de redes de alta velocidad, así como equipos de cómputo —como grandes sistemas de almacenamiento en disco y cinta.

En una situación de diseño puede elegirse un CPLD siempre que el circuito necesario no sea muy grande o cuando el dispositivo deba realizar su función en seguida de la aplicación de potencia al circuito. Los FPGA no son buenos para este último caso porque, como ya se señaló, están configurados mediante elementos de almacenamiento volátiles que pierden el contenido que guardan cuando la energía se desconecta. Esta propiedad resulta en un retraso antes de que el chip FPGA pueda cumplir su función cuando se enciende.

Los FPGA son adecuados para la implementación de circuitos de una gran variedad de tamaños, desde cerca de 1 000 hasta más de un millón de compuertas lógicas equivalentes. Además del tamaño, el diseñador ha de considerar otros criterios, como la velocidad de operación necesaria del circuito, las restricciones de disipación de potencia y el costo de los chips. Cuando los FPGA no satisfacen uno o más de los requisitos, el usuario puede optar por crear un chip fabricado a la medida, como se describe a continuación.

3.7 CHIPS DISEÑADOS A LA MEDIDA, CELDAS ESTÁNDAR Y ARREGLOS DE COMPUERTAS

El factor principal que limita el tamaño de un circuito que puede acomodarse en un PLD es la existencia de interruptores programables. Aunque éstos ofrecen el importante beneficio de ser programables por el usuario, ocupan una cantidad significativa de espacio en el chip, lo que desemboca en mayores costos. También resultan en una reducción de la rapidez de operación de los circuitos y un aumento del consumo de potencia. En esta sección expondremos algunas tecnologías de circuitos integrados que no contienen interruptores programables.

Para lograr el número más grande de compuertas lógicas, la mayor rapidez del circuito o la menor potencia es posible fabricar un *chip a la medida*. Mientras que un PLD es prefabricado y contiene compuertas lógicas e interruptores programables que se programan para realizar el circuito de un usuario, un chip a la medida se crea desde cero. El diseñador de un chip a la medida goza de completa flexibilidad para decidir el tamaño del chip, el número de transistores que tendrá, la ubicación de cada uno de ellos y la forma en la que estarán conectados. El proceso de definir con exactitud dónde se situará cada transistor y cable en el chip se llama *diagrama de conexión de chip*. Los chips a la medida requieren mucho trabajo de diseño y, por tanto, son costosos. En consecuencia, se producen sólo cuando las partes estándar como los FPGA no satisfacen las necesidades. Para justificar el gasto en un chip a la medida cabe esperar que el producto por diseñar se venda en cantidades suficientes para recuperar el costo. Dos ejemplos de productos que se realizan con chips a la medida son los microprocesadores y los chips de memoria.

En situaciones en que el diseñador del chip no necesita flexibilidad total para hacer el diagrama de conexión de cada transistor de un chip a la medida, parte del esfuerzo de diseño puede obviarse mediante una tecnología conocida como *celdas estándar*. Los chips hechos con esta tecnología se denominan *circuitos integrados de aplicación específica* (ASIC, *application-specific integrated circuits*). Esta tecnología se ilustra en la figura 3.40, donde se describe una pequeña parte de un chip. Las filas de las compuertas lógicas pueden conectarse mediante cables creados en los canales de enrutamiento que hay entre las filas de compuertas. En general, en un chip de tales características es posible emplear muchos tipos de compuertas lógicas. Las compuertas disponibles son preconstruidas y se almacenan en una biblioteca a la que el diseñador puede acceder. En la figura 3.40 los cables se dibujan en negro y gris oscuro. Se usa este esquema porque los cables metálicos pueden crearse sobre circuitos integrados en múltiples *capas*, lo que permite que dos cables se crucen sin crear un cortocircuito. Los cables grises representan una capa de cables metálicos, en tanto que los negros son una capa diferente. Cada cuadrado gris representa una conexión de cableado inalterable (llamada *vía*) entre un cable de una capa y otro de otra. En la tecnología actual es posible tener ocho o más capas de cables metálicos.

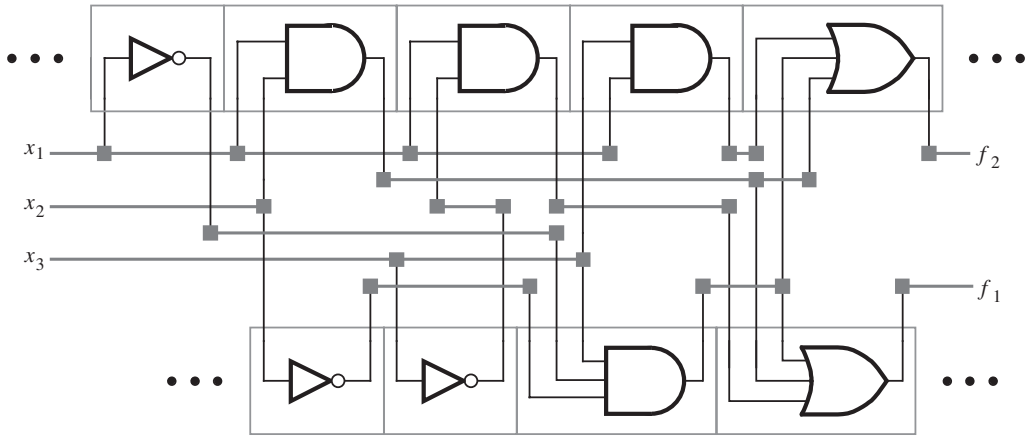


Figura 3.40 Sección de dos filas en un chip de celda estándar.

Algunas capas metálicas pueden colocarse en la parte superior de los transistores en las compuertas lógicas, lo que resulta en un diagrama de conexión de chip más eficiente.

Igual que un chip a la medida, uno de celda estándar se crea desde cero de acuerdo con las especificaciones del usuario. Los circuitos que se muestran en la figura 3.40 implementan las dos funciones lógicas realizadas en el PLA de la figura 3.26: $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$ y $f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Debido al costo, un chip de celda estándar nunca se crearía para un circuito pequeño como éste y, por tanto, la figura sólo describe una parte de un chip mucho más grande. El diagrama de conexiones de cada compuerta (celdas estándar) es prediseñado y fijo. El diagrama de conexiones del chip puede crearse automáticamente mediante herramientas CAD gracias al ordenamiento regular de las compuertas lógicas (celdas) en filas. Un chip típico tiene filas muy largas de compuertas lógicas con un gran número de cables ente cada par de filas. Los bloques de I/O alrededor de la periferia conectan los pines del paquete de chip, que casi siempre es un QFP, PGA o BGA.

Otra tecnología, similar a las celdas estándar, es la de *arreglo de compuerta*, en la que ciertas partes del chip son prefabricadas y otras se fabrican a la medida para un circuito de usuario en particular. Este concepto explota el hecho de que los circuitos integrados se fabrican en una secuencia de pasos, algunos de ellos para crear transistores y otros para crear alambres que conecten los transistores. En la tecnología de arreglo de compuerta, el fabricante realiza la mayor parte de los pasos de fabricación, por lo general los de la creación de los transistores, sin considerar los requisitos del circuito del usuario. Este proceso resulta en una oblea de silicio (véase la figura 1.1) de chips parcialmente terminados, llamada *plantilla de arreglo de compuerta*. Más tarde, la plantilla se modifica, casi siempre mediante la fabricación de cables que conecten los transistores, para crear el circuito del usuario en cada chip terminado. El enfoque de arreglo de compuerta ofrece ahorros de costo en comparación con el de chip a la medida porque el fabricante puede amortizar el costo de la fabricación del chip con un gran número de obleas-plantilla, las cuales son idénticas. Hay muchas variantes de la tecnología de arreglo de compuerta. Algunas tienen celdas lógicas relativamente grandes, mientras que otras son configurables aun en el nivel de un solo transistor.

En la figura 3.41 se presenta un ejemplo de plantilla de arreglo de compuerta. El arreglo de compuertas contiene un arreglo bidimensional de celdas lógicas. La estructura general del chip es similar a la de uno de celda estándar, salvo que en el arreglo de compuertas todas las celdas lógicas son idénticas. Aunque los tipos de celdas lógicas usadas en los arreglos de compuertas varían, un ejemplo común es una compuerta NAND de dos o tres entradas. En algunos arreglos de compuertas existen espacios vacíos entre las filas de celdas lógicas a fin de acomodar los cables que se agregarán después para conectarlas. Sin embargo, la mayoría de los arreglos de compuertas no tiene tales espacios, y los cables de interconexión se fabrican encima de las celdas lógicas. Este diseño es posible porque, como indicamos para la figura 3.40, los cables metálicos pueden crearse sobre un chip en varias capas. Este enfoque se conoce como *tecnología de mar de compuertas*. En la figura 3.42 se muestra una pequeña sección de un arreglo de compuertas que se ha adaptado a la medida para implementar la función lógica $f = x_2\bar{x}_3 + x_1x_3$. Como se indica en la sección 2.7, es fácil comprobar que este circuito sólo con compuertas NAND equivale a la forma AND-OR del circuito.

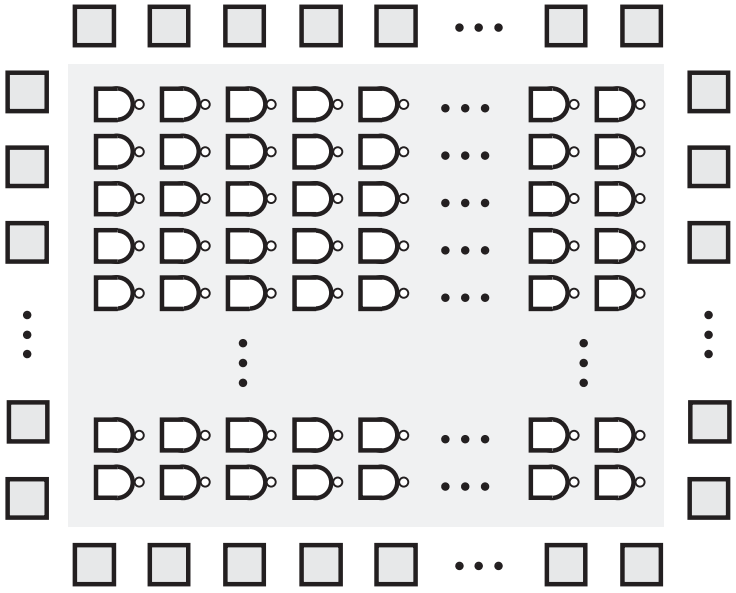


Figura 3.41 Arreglo de compuerta en mar de compuertas.

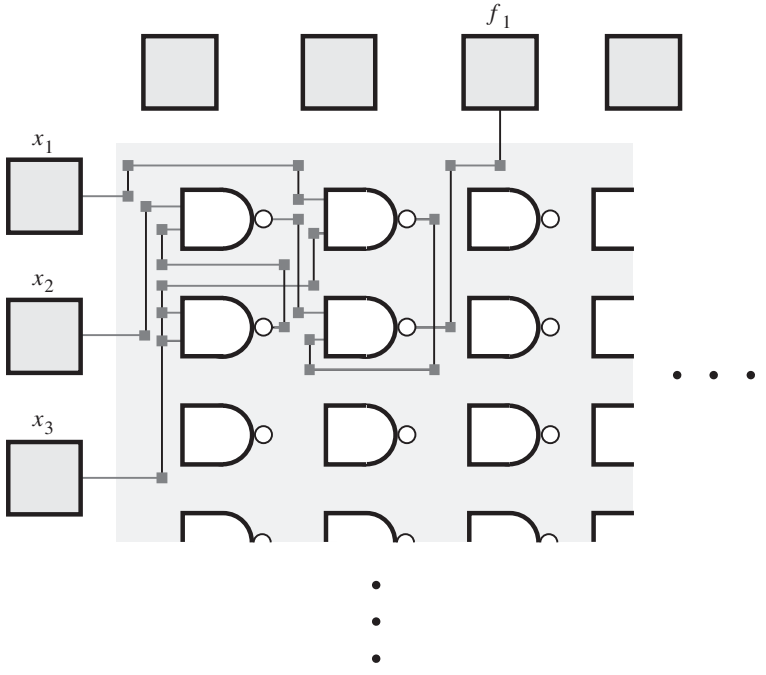


Figura 3.42 La función lógica $f_1 = x_2\overline{x}_3 + x_1x_3$ del arreglo de compuerta de la figura 3.41.

3.8 ASPECTOS PRÁCTICOS

Hasta ahora hemos descrito la operación básica de los circuitos de compuertas lógicas y dado ejemplos de chips comerciales. En esta sección proporcionamos información detallada de varios aspectos de los circuitos digitales. Describiremos cómo se fabrican en silicio los transistores y brindaremos una explicación detallada de cómo operan éstos. Abordaremos las ventajas de los circuitos lógicos y expondremos aspectos significativos de los retrasos en la propagación de señales y la disipación de potencia en las compuertas lógicas.

3.8.1 FABRICACIÓN Y COMPORTAMIENTO DE LOS MOSFET

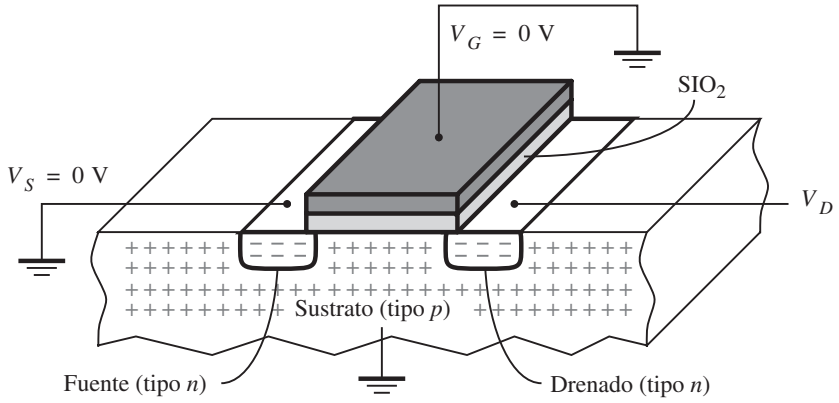
Para entender la operación de los transistores NMOS y PMOS es preciso considerar cómo se construyen en un circuito integrado. Los circuitos integrados se fabrican en obleas de silicio. Una oblea de silicio (véase la figura 1.1) mide 6, 8 o 12 pulgadas de diámetro y en cierto modo es similar a un disco compacto (CD). En una oblea se fabrican muchos circuitos integrados; después la oblea se corta para ofrecer los chips individuales.

El silicio es un *semiconductor eléctrico*, lo que significa que se le puede manipular de tal modo que a veces conduce corriente eléctrica y a veces no. Un transistor se fabrica creando áreas en el sustrato de silicio que tengan exceso de carga eléctrica positiva o negativa. Las áreas cargadas negativamente se llaman *tipo n* y las cargadas positivamente, *tipo p*. En la figura 3.43 se ilustra la estructura de un transistor NMOS. Tiene silicio tipo *n* para las terminales de fuente y drenado, y tipo *p* para la de sustrato. Los cables de metal se usan para hacer las conexiones eléctricas a las terminales de la fuente y el drenado.

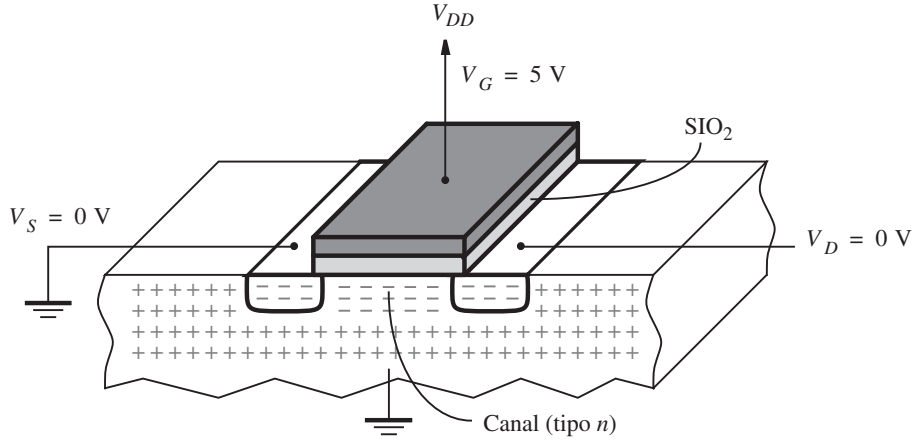
Cuando se inventaron los MOSFET, la terminal compuerta se hizo de metal. Ahora se utiliza un material conocido como *polisilicio*. Igual que un metal, el polisilicio es conductor, pero es preferible aquél porque tiene propiedades que permiten fabricar los MOSFET con dimensiones en extremo pequeñas. La compuerta se aísla eléctricamente del resto de los transistores mediante una capa de dióxido de silicio (SiO_2), que es un tipo de vidrio que actúa como aislador eléctrico entre la terminal compuerta y el sustrato del transistor. La operación del transistor está regida por campos eléctricos generados por medio de voltaje aplicado a sus terminales, como se explica a continuación.

En la figura 3.43 los niveles de voltaje aplicados a las terminales fuente, compuerta y drenado se etiquetan V_S, V_G y V_D, respectivamente. Considérese primero la situación presentada en la figura 3.43*a*, en la que tanto la fuente como la compuerta se conectan a *Gnd* ($V_S = V_G = 0$ V). La fuente tipo *n* y el drenado tipo *n* se aíslan uno de otro mediante el sustrato tipo *p*. En términos eléctricos, hay dos diodos entre la fuente y el drenado. Mediante la unión *p-n* entre el sustrato y la fuente se forma un diodo, y el otro diodo se forma mediante la unión *p-n* entre el sustrato y el drenado. Estos diodos espalda con espalda representan una muy alta resistencia (aproximadamente 10^{12} Ω ([1]) entre el drenado y la fuente que evita flujo de corriente. Se dice que el transistor está *apagado* o *cortado*, en este estado.

Ahora considérese el efecto de aumentar el voltaje en la terminal compuerta respecto al de la fuente. Sea V_{GS} el voltaje de compuerta a fuente. Si V_{GS} es mayor que cierto voltaje positivo mínimo, llamado *voltaje umbral* V_T, entonces el transistor cambia de un interruptor abierto a un interruptor cerrado, como explicamos en seguida. El nivel exacto de V_T depende de muchos factores, pero casi siempre se aproxima a 0.2 V_{DD}.



a) Cuando V_{GS} = 0 V, el transistor está apagado



b) Cuando V_{GS} = 5 V, el transistor está encendido

Figura 3.43 Estructura física de un transistor NMOS.

El estado del transistor cuando $V_{GS} > V_T$ se ilustra en la figura 3.43*b*. La terminal compuerta está conectada a V_{DD}, lo que resulta en $V_{GS} = 5$ V. El voltaje positivo en la compuerta atrae los electrones libres que salen en la terminal fuente tipo *n*, así como en otras áreas del transistor, hacia la compuerta. Puesto que los electrones no pueden pasar por la capa de vidrio bajo la compuerta, se juntan en la región del sustrato entre la fuente y el drenado, que se llama *canal*. Esta concentración de electrones *invierte* el silicio en el área del canal del tipo *p* al tipo *n*, que efectivamente conecta la fuente y el drenado. El tamaño del canal se determina por la longitud y el ancho de la compuerta. La longitud del canal *L* es la dimensión de la compuerta entre la fuente y el drenado, y el ancho del canal *W* es la otra dimensión. También puede pensarse que

el canal tiene una profundidad, que depende de los voltajes aplicados a la fuente, la compuerta y el drenado.

No puede pasar corriente por el nodo compuerta del transistor debido a la capa de vidrio que la aísla del sustrato. Una corriente I_D puede fluir del nodo drenado a la fuente. Para un valor fijo de $V_{GS} > V_T$, el valor de I_D depende del voltaje aplicado al canal V_{DS}. Si $V_{DS} = 0$ V, entonces no fluye corriente. Conforme V_{DS} crece, I_D aumenta de manera casi lineal con el V_{DS} aplicado, en tanto V_D sea suficientemente pequeño como para proporcionar al menos V_T voltios a través del extremo drenado del canal, es decir: $V_{GD} > V_T$. En este rango de voltajes $0 < V_{DS} < (V_{GS} - V_T)$, se dice que el transistor opera en la *región de tríodo*, también llamada *región lineal*. La relación entre voltaje y corriente se aproxima con la ecuación

$$I_D = k'_n \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \qquad \textbf{[3.1]}$$

El símbolo k'_n se denomina *parámetro de transconductancia del proceso*. Se trata de una constante que depende de la tecnología empleada y tiene las unidades A/V^2.

Conforme V_D aumenta, el flujo de corriente por el transistor aumenta, como está dado por la ecuación 3.1, pero sólo hasta cierto punto. Cuando $V_{DS} = V_{GS} - V_T$, la corriente alcanza su valor máximo. Para valores más grandes de V_{DS}, el transistor ya no opera en la región de tríodo. Puesto que la corriente se halla en su valor saturado (máximo), se dice que el transistor está en la región de saturación. Ahora la corriente es independiente de V_{DS} y está dada por la expresión

$$I_D = \frac{1}{2}k'_n \frac{W}{L}(V_{GS} - V_T)^2 \qquad \textbf{[3.2]}$$

En la figura 3.44 se muestra la forma de la relación corriente-voltaje en el transistor NMOS para un valor fijo de $V_{GS} > V_T$. En la figura se indica el punto en el que el transistor deja la región de tríodo y entra en la de saturación, lo cual ocurre en $V_{DS} = V_{GS} - V_T$.

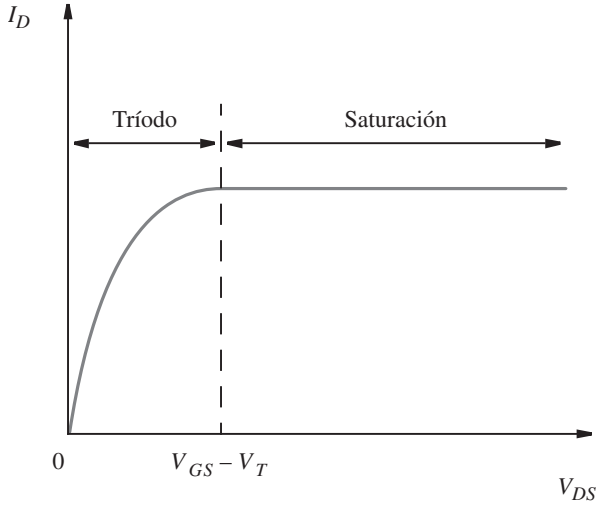


Figura 3.44 Relación voltaje-corriente en el transistor NMOS.

Ejemplo 3.3

Suponga los valores $k_n' = 60\ \mu A/V^2$, $W/L = 2.0\ \mu m/0.5\ \mu m$, $V_S = 0$ V, $V_G = 5$ V, la corriente en el transistor está dada por la ecuación 3.1 como $I_D \approx 1.7$ mA. Si $V_D = 5$ V, la corriente de saturación se calcula con la ecuación 3.2 como $I_D \approx 2$ mA.

El transistor PMOS

El comportamiento de los transistores PMOS es el mismo que el de los NMOS, excepto que todos los voltajes y corrientes están invertidos. La terminal fuente del transistor PMOS es la terminal con el nivel de voltaje más alto (recuérdese que para un transistor NMOS la terminal fuente es la que tiene el nivel de voltaje más bajo), y el voltaje umbral requerido para encender el transistor tiene un valor negativo. Los transistores PMOS tienen la misma construcción física que los NMOS, salvo que donde el transistor NMOS tiene silicio tipo *n*, el PMOS tiene tipo *p*, y viceversa. Para un transistor PMOS, el equivalente de la figura 3.43*a* es conectar los nodos fuente y compuerta a V_{DD}, caso en el que el transistor se apaga. A fin de encender el transistor PMOS, equivalente a la figura 3.43*b*, el nodo compuerta se establecería en *Gnd*, lo que resulta en $V_{GS} = -5$ V.

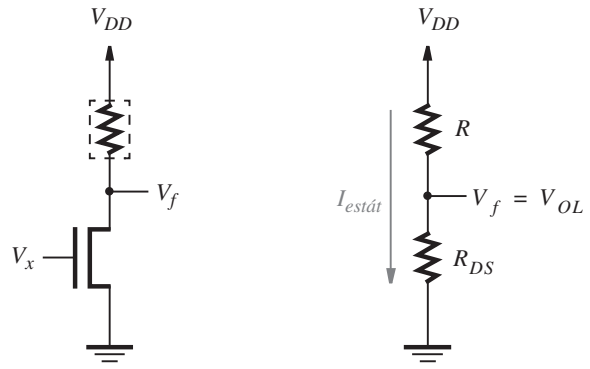
Como el canal es silicio tipo *p* y no tipo *n*, el mecanismo físico para la conducción de corriente en los transistores PMOS es diferente del de los transistores NMOS. Una explicación detallada de este tema va más allá del alcance de esta obra, pero es preciso mencionar una implicación. Las ecuaciones 3.1 y 3.2 usan el parámetro k_n'. El parámetro correspondiente para un transistor PMOS es k_p', pero la corriente fluye con mayor facilidad en el silicio tipo *n* que en el tipo *p*, con el resultado de que en una tecnología típica, $k_p' \approx 0.4 \times k_n'$. Para que un transistor PMOS tenga capacidad de corriente igual a la de un transistor NMOS hay que usar *W*/*L* aproximadamente dos o tres veces mayor en el transistor PMOS. En las compuertas lógicas los tamaños de los transistores NMOS y PMOS suelen elegirse tomando en cuenta este factor.

3.8.2 MOSFET CON RESISTENCIA DE ENCENDIDO (ON-RESISTANCE)

En la sección 3.1 consideramos los MOSFET como interruptores ideales que tienen resistencia infinita cuando se apagan y resistencia cero cuando se encienden. La resistencia real del canal cuando el transistor se enciende, la cual recibe el nombre de *on-resistance*, está dada por V_{DS}/I_D. Con la ecuación 3.1 puede calcularse la on-resistance en la región de tríodo, como se muestra en el ejemplo 3.4.

Ejemplo 3.4

Considérese un inversor CMOS en el que el voltaje de entrada V_x es igual a 5 V. El transistor NMOS se enciende y el voltaje de salida V_f está cerca de 0 V. Por tanto, V_{DS} para el transistor NMOS está cerca de cero y éste opera en la región de tríodo. En la curva de la figura 3.44, el transistor funciona en un punto muy cercano al origen. Aunque el valor de V_{DS} es pequeño, no es exactamente cero. En la sección siguiente se explica que V_{DS} suele aproximarse a 0.1 mV. En consecuencia, la corriente I_D no es exactamente cero; se define por la ecuación 3.1. En esta ecuación puede des-



a) Compuerta NOT NMOS *b*) V_x = 5 V

Figura 3.45 Niveles de voltaje en el inversor NMOS.

preciarse el término que involucra V_{DS}^2 porque V_{DS} es pequeño. En este caso la on-resistance se aproxima mediante

$$R_{DS} = V_{DS}/I_D = 1/\left[k_n' \frac{W}{L}(V_{GS} - V_T)\right] \qquad \textbf{[3.3]}$$

Si suponemos los valores $k_n' = 60\ \mu\text{A/V}^2$, $W/L = 2.0\ \mu\text{m}/0.5\ \mu\text{m}$, $V_{GS} = 5$ V, y $V_T = 1$ V, se obtiene $R_{DS} \approx 1\ \text{k}\Omega$.

3.8.3 NIVELES DE VOLTAJE EN COMPUERTAS LÓGICAS

En la figura 3.1 mostramos que los valores lógicos se representan mediante límites de niveles de voltaje. Ahora prestaremos mayor atención al tema de los niveles de voltaje.

Los niveles de voltaje alto y bajo en una familia lógica se caracterizan por medio de la operación de su inversor básico. En la figura 3.45*a* se reproduce el circuito de la figura 3.5 para un inversor construido con tecnología NMOS. Cuando $V_x = 0$ V, el transistor NMOS se apaga. No fluye corriente; por ende, $V_f = 5$ V. Cuando $V_x = V_{DD}$, el transistor NMOS se enciende. Para calcular el valor de V_f, el transistor NMOS puede representarse mediante un resistor con un valor R_{DS}, como se ilustra en la figura 3.45*b*. Entonces V_f está dado por el divisor de voltaje

$$V_f = V_{DD}\frac{R_{DS}}{R_{DS} + R}$$

Ejemplo 3.5 Supóngase que $R = 25\ \text{k}\Omega$. Con el resultado del ejemplo 3.4, $R_{DS} = 1\ \text{k}\Omega$, que produce $V_f \approx 0.2$ V.

Como se indicó en la figura 3.45*b*, una corriente $I_{estát}$ fluye a través del inversor NMOS bajo la condición estática $V_x = V_{DD}$. Esta corriente está dada por

$$I_{estát} = V_f / R_{DS} = 0.2 \text{ V} / 1 \text{ k}\Omega = 0.2 \text{ mA}$$

Esta corriente estática tiene implicaciones importantes, que abordaremos en la sección 3.8.6.

En los modernos circuitos NMOS, el dispositivo de subida *R* se implementa usando un transistor PMOS. Tales circuitos se conocen como *circuitos seudoNMOS*. Son completamente compatibles con los circuitos CMOS; por tanto, un solo chip puede contener compuertas tanto CMOS como seudoNMOS. En el ejemplo 3.13 se muestra el circuito de un inversor seudo NMOS y se analiza cómo calcular sus niveles de voltaje de salida.

El inversor CMOS

Es habitual usar los símbolos V_{OH} y V_{OL} para caracterizar los niveles de voltaje de un circuito lógico. El significado de V_{OH} es el voltaje producido cuando la salida es alta (*output high*). De manera similar, V_{OL} se refiere al voltaje que se produce cuando la salida es baja (*output low*). Como ya vimos, en el inversor NMOS $V_{OH} = V_{DD}$, y V_{OL} se acerca a 0.2 V.

Considérese de nuevo el inversor CMOS de la figura 3.12*a*. Su relación de voltaje salida-entrada se resume mediante la *característica de transferencia de voltaje* mostrada en la figura 3.46. La curva proporciona el valor de estado estacionario de V_f para cada valor de V_x. Cuando $V_x = 0$ V, el transistor NMOS se apaga. No fluye corriente; por ende, $V_f = V_{OH} = V_{DD}$. Cuando $V_x = V_{DD}$, el transistor PMOS se apaga, no fluye corriente y $V_f = V_{OL} = 0$ V. Para redondear, cabe decir que, aun cuando un transistor esté apagado, por él puede fluir una pequeña corriente, llamada corriente de *fuga*. Esta corriente tiene un ligero efecto en V_{OH} y V_{OL}. Por ejemplo, un valor típico de V_{OL} es 0.1 mV, en lugar de 0 V [1].

En la figura 3.46 se incluyen etiquetas en los puntos donde el voltaje de salida comienza a cambiar de alto a bajo, y viceversa. El voltaje V_{IL} representa el punto donde el voltaje de salida es alto y la pendiente de la curva es igual a –1. Este nivel de voltaje se define como el máximo nivel de voltaje de entrada que el inversor interpretará como bajo, y por ende produce una salida alta. De manera similar, el voltaje V_{IH}, que es el otro punto de la curva donde la pendiente es igual a –1, es el mínimo nivel de voltaje de entrada que el inversor interpretará como alto, y por ende produce una salida baja. Los parámetros V_{OH}, V_{OL}, V_{IL} y V_{IH} son importantes para cuantificar la robustez de una familia lógica, como explicaremos en seguida.

3.8.4 MARGEN DE RUIDO

Considérense las dos compuertas NOT que aparecen en la figura 3.47*a*. Sean las compuertas a la izquierda y derecha N_1 y N_2, respectivamente. Los circuitos electrónicos están constantemente sujetos a perturbaciones aleatorias, llamadas ruido, que pueden alterar los niveles de voltaje de salida producidos por la compuerta N_1. Es indispensable que este ruido no haga que la compuerta N_2 malinterprete un valor lógico bajo como uno alto, y viceversa. Considérese el caso en el que N_1 produce su nivel de voltaje bajo V_{OL}. La presencia de ruido puede alterar el nivel de voltaje, pero en tanto permanezca menor que V_{IL}, N_2 lo interpretará correctamente. La capacidad para

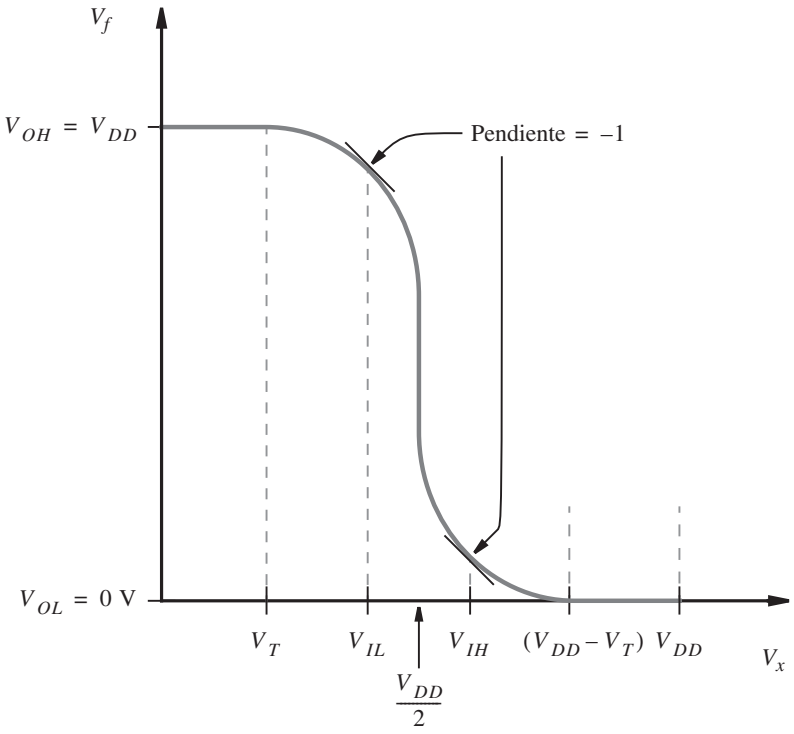


Figura 3.46 Característica de transferencia de voltaje para el inversor CMOS.

tolerar ruido sin afectar la operación correcta del circuito se conoce como margen de ruido. Para el voltaje de salida bajo, el margen de ruido bajo se define como

$$NM_L = V_{IL} - V_{OL}$$

Una situación similar ocurre cuando N_1 produce su voltaje de salida alto V_{OH}. Cualquier ruido existente en el circuito puede alterar el nivel de voltaje, pero N_2 lo interpretará correctamente en tanto el voltaje sea mayor que V_{IH}. El margen de ruido alto se define como

$$NM_H = V_{OH} - V_{IH}$$

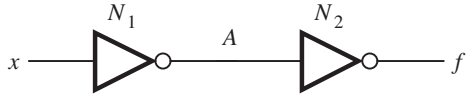
Ejemplo 3.6 Para un tecnología dada, la característica de transferencia de voltaje del inversor básico determina los niveles V_{OH}, V_{OL}, V_{IL} y V_{IH}. Para el CMOS mostrado en la figura 3.46, $V_{OH} = V_{DD}$ y $V_{OL} = 0$ V. Al encontrar los dos puntos donde la pendiente de la característica de transferencia de voltaje es igual a –1 puede demostrarse [1] que $V_{IL} \cong \frac{1}{8}(3V_{DD} + 2V_T)$ y $V_{IH} \cong \frac{1}{8}(5V_{DD} - 2V_T)$. Para el valor típico $V_T = 0.2\ V_{DD}$, esto produce

$$NM_L = NM_H = 0.425 \times V_{DD}$$

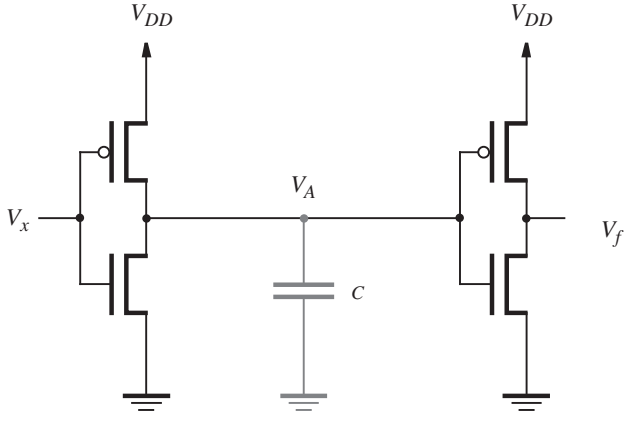
Por tanto, el margen de ruido disponible depende del nivel de voltaje de la fuente de poder. Para $V_{DD} = 5$ V, el margen de ruido es 2.1 V, y para $V_{DD} = 3.3$ V, el margen de ruido es 1.4 V.

3.8.5 OPERACIÓN DINÁMICA DE LAS COMPUERTAS LÓGICAS

En la figura 3.47*a*, el nodo entre las dos compuertas se etiqueta con *A*. Debido a la forma en la que los transistores se construyen en silicio, N_2 tiene el efecto de contribuir a una carga capacitiva en el nodo *A*. En la figura 3.43 se muestra que los transistores se construyen con varias capas de diferentes materiales. Siempre que dos tipos de material se encuentren o traslapen dentro del transistor puede crearse efectivamente un capacitor. Esta capacitancia se llama *capacitancia parásita* porque es un efecto colateral indeseable de la fabricación de transistores. En la figura 3.47 el interés se centra en la capacitancia que hay en el nodo *A*. Varios capacitores parásitos se unen a este nodo, algunos causados por N_1 y otros por N_2. Entre la entrada del inversor N_2 y tierra existe un capacitor parásito significativo. El valor de este capacitor depende del tamaño de los transistores de N_2. Cada transistor aporta una *capacitancia de compuerta*, $C_g = W \times L \times C_{ox}$. El parámetro C_{ox}, llamado *capacitancia de óxido*, es una constante para la tecnología que se use y tiene las unidades fF/μm^2. Una capacitancia adicional la causan los transistores de N_1 y también los cables de metal que se unen al nodo *A*. Es posible representar toda la capacitancia parásita



a) Compuerta NOT que alimenta otra compuerta NOT



b) Carga capacitiva en el nodo A

Figura 3.47 Capacitancia parásita en circuitos integrados.

por medio de una sola capacitancia equivalente entre el nodo *A* y tierra [2]. En la figura 3.47*b* esta capacitancia equivalente se etiqueta con *C*.

La existencia de la capacitancia parásita tiene un efecto negativo en la velocidad de operación de los circuitos lógicos. El voltaje a través de un capacitor no puede cambiar de forma instantánea. El tiempo necesario para carga o descarga de un capacitor depende del tamaño de la capacitancia *C* y de la cantidad de corriente que pasa por él. En el circuito de la figura 3.47*b*, cuando el transistor PMOS de N_1 se enciende, el capacitor se carga a V_{DD}; se descarga cuando el transistor NMOS se enciende. En cada caso, el flujo de corriente I_D a través del transistor involucrado y el valor de *C* determinan la tasa de carga y descarga del capacitor.

En el capítulo 2 expusimos el concepto de diagrama de tiempo, y en la figura 2.10 presentamos un diagrama de tiempo en el que las formas de onda tienen bordes perfectamente verticales en la transición de un nivel lógico al otro. En circuitos reales, las formas de onda no tienen esta forma "ideal", sino que más bien tienen la apariencia de los mostrados en la figura 3.48. En ella se presenta una forma de onda para la entrada V_x de la figura 3.47*b* y se muestra la forma de onda resultante en el nodo *A*. Suponemos que V_x inicialmente está en el nivel de voltaje V_{DD} y luego realiza un transición a 0. Una vez que V_x alcanza un voltaje suficientemente bajo, N_1 empieza a conducir voltaje V_A hacia V_{DD}. A causa de la capacitancia parásita, V_A no puede cambiar instantáneamente y resulta una forma de onda con la forma indicada en la figura. El tiempo necesario para que V_A cambie de bajo a alto se denomina *tiempo de elevación*, t_r, que se define como el tiempo transcurrido desde el instante en que V_A se halla a 10% de V_{DD}, hasta que alcanza 90%. En la figura 3.48 también se define la cantidad total de tiempo necesario para que el cambio en V_x cause un cambio en V_A. Este intervalo se denomina *retraso de propagación* —que a menudo se escribe t_p— del inversor. Se trata del tiempo desde que V_x alcanza 50% de V_{DD} hasta que V_A llega al mismo nivel.

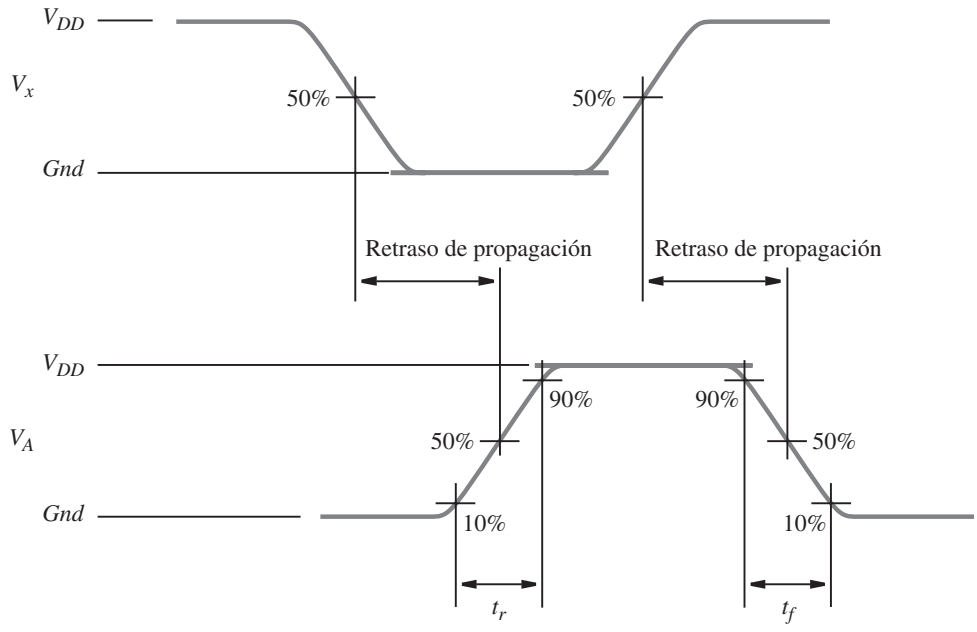


Figura 3.48 Formas de onda de voltaje para compuertas lógicas.

Después de permanecer en 0 V durante algún tiempo, V_x cambia de vuelta a V_{DD}, lo que ocasiona que N_1 descargue C a *Gnd*. En este caso el tiempo de transición en el nodo A pertenece a un cambio de alto a bajo, al que se le conoce como *tiempo de caída*, t_f, de 90% de V_{DD} a 10% de V_{DD}. Como se indica en la figura, hay un retraso de propagación correspondiente para que el nuevo cambio en V_x afecte a V_A. En una compuerta lógica, los tamaños relativos de los transistores PMOS y NMOS se eligen de modo que t_r y t_f tengan casi el mismo valor.

Las ecuaciones 3.1 y 3.2 especifican la cantidad de flujo de corriente a través de un transistor NMOS. Dado el valor de C en la figura 3.47, es posible calcular el retraso de propagación para un cambio en V_A de alto a bajo. Por simplicidad, supóngase que V_x inicialmente es 0 V; por ende, el transistor PMOS se enciende y $V_A = 5$ V. Entonces V_x cambia a V_{DD} en el tiempo 0, lo que ocasiona que el transistor PMOS se apague y el NMOS se encienda. Entonces el retraso de propagación es el tiempo que se requiere para que C se descargue a través del transistor NMOS al voltaje $V_{DD}/2$. Cuando V_x cambia primero a V_{DD}, $V_A = 5$ V; por tanto, el transistor NMOS tendrá $V_{DS} = V_{DD}$ y estará en la región de saturación. La corriente I_D está dada por la ecuación 3.2. Una vez que V_A cae por abajo de $V_{DD} - V_T$, el transistor NMOS entrará a la región de tríodo donde I_D está dada por la ecuación 3.1. Para los propósitos del texto, es posible aproximar el flujo de corriente mientras V_A cambia de V_{DD} a $V_{DD}/2$ al encontrar el promedio de los valores dados por la ecuación 3.2 con $V_{DS} = V_{DD}$ y la ecuación 3.1 con $V_{DS} = V_{DD}/2$. Si se usa la expresión básica para el tiempo necesario para cargar un capacitor (véase ejemplo 3.11), se tiene

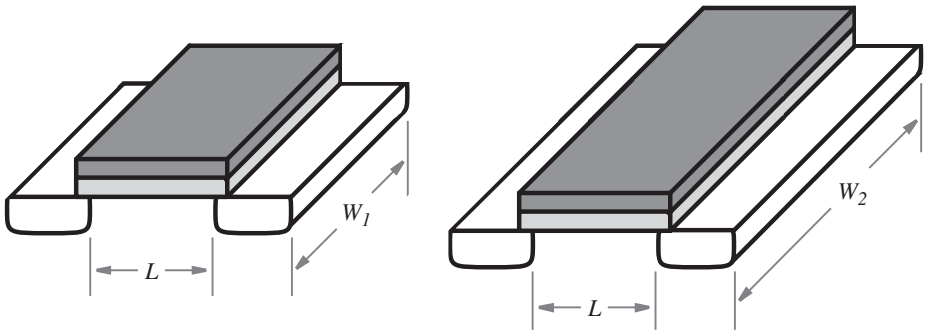
$$t_p = \frac{C\Delta V}{I_D} = \frac{CV_{DD}/2}{I_D}$$

La sustitución del valor promedio de I_D explicado líneas arriba produce [1]

$$t_p \cong \frac{1.7\,C}{k_n' \frac{W}{L} V_{DD}} \qquad \textbf{[3.4]}$$

Esta expresión indica que la velocidad del circuito depende tanto del valor de C como de las dimensiones del transistor. El retraso puede reducirse haciendo C más pequeño o la razón W/L más grande. La expresión muestra el tiempo de propagación cuando la salida cambia de un nivel alto a uno bajo. El tiempo de propagación de bajo a alto está dado por la misma expresión pero usando k_p' y W/L del transistor PMOS.

En los circuitos lógicos, L usualmente se fija en el valor mínimo permitido de acuerdo con las especificaciones de la tecnología de fabricación utilizada. El valor de W se elige según la cantidad de flujo de corriente, y por tanto de retraso de propagación, que se desea. En la figura 3.49 se ilustran dos tamaños de transistores. En el inciso *a*) se describe un transistor de tamaño mínimo, que se usaría en un circuito donde la carga capacitiva fuera pequeña o la velocidad de operación no fuese crucial. En la figura 3.49*b* se muestra un transistor más grande, de la misma longitud que el transistor del inciso *a*) pero de mayor ancho. Siempre hay algo que se pierde y algo que se gana al elegir el tamaño de un transistor, pues uno grande ocupa más espacio en un chip que uno pequeño. Además, al aumentar W no sólo crece la cantidad de flujo de corriente en el transistor, sino también la capacitancia parásita (recuérdese que la capacitancia C_g entre la terminal compuerta y tierra es proporcional a $W \times L$), que tiende a contrarrestar algunas de las mejoras esperadas en el rendimiento. En los circuitos lógicos se utilizan grandes transistores donde deben conducirse cargas capacitivas altas y donde los retrasos de propagación de señal han de minimizarse.



a) Transistor pequeño *b*) Transistor más grande

Figura 3.49 Tamaños de transistor.

Ejemplo 3.7 En el circuito de la figura 3.47, supóngase que $C = 70$ fF y que $W/L = 2.0\ \mu$m/0.5 μm. Además, $k_n' = 60\ \mu\text{A/V}^2$ y $V_{DD} = 5$ V. Con la ecuación 3.4, el retraso de propagación de alto a bajo del inversor es $t_p \approx 0.1$ ns.

3.8.6 DISIPACIÓN DE POTENCIA EN LAS COMPUERTAS LÓGICAS

En un circuito electrónico es importante considerar la cantidad de potencia eléctrica consumida por los transistores. La tecnología de circuitos integrados permite la fabricación de millones de transistores en un solo chip; por tanto, la cantidad de potencia que utiliza un solo transistor debe ser pequeña. La disipación de potencia es una consideración importante en todas las aplicaciones de los circuitos lógicos, pero resulta crucial cuando se trata de situaciones relacionadas con equipo que opera con baterías, como las computadoras portátiles.

Considérese de nuevo el inversor NMOS de la figura 3.45. Cuando $V_x = 0$, no fluye corriente y por tanto no se usa potencia. Pero cuando $V_x = 5$ V, se consume potencia debido a la corriente $I_{estát}$. La potencia consumida está dada por $P_S = I_{estát} V_{DD}$. En el ejemplo 3.5 calculamos $I_{estát} = 0.2$ mA. Entonces, la potencia consumida es $P_S = 0.2\ \text{mA} \times 5\ \text{V} = 1.0$ mW. Si suponemos que un chip contiene, digamos, el equivalente de 10 000 inversores, entonces el consumo de potencia total ¡es de 10 W! Debido a este gran consumo de potencia, las compuertas estilo NMOS sólo se usan en aplicaciones de propósito especial, tema que estudiaremos en la sección 3.8.8.

Para distinguir entre la potencia consumida durante condiciones de estado estacionario y la consumida cuando las señales varían suelen definirse dos tipos de potencia. La *potencia estática* es disipada por la corriente que fluye en el estado estacionario, y la *potencia dinámica* se consume cuando la corriente fluye debido a cambios en los niveles de señal. Los circuitos NMOS consumen tanto potencia estática como dinámica, mientras que los circuitos CMOS sólo consumen potencia dinámica.

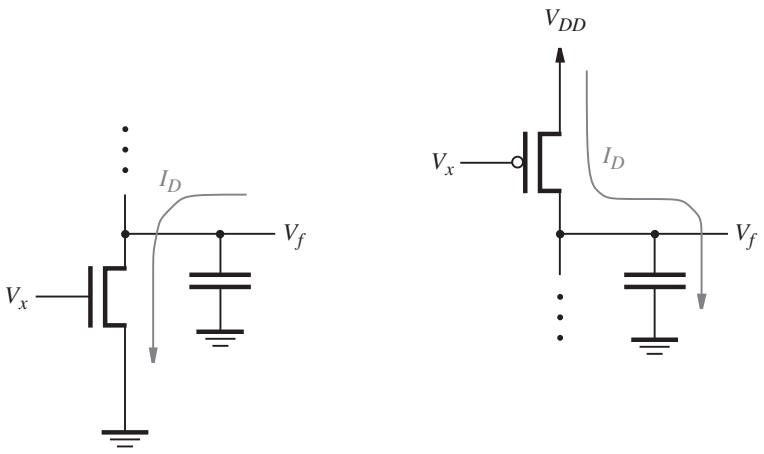
Considérese el inversor CMOS presentado en la figura 3.12*a*. Cuando la entrada V_x es baja no fluye corriente porque el transistor NMOS está apagado. Cuando V_x es alta el transistor PMOS

está apagado y, de nuevo, no pasa corriente. Por tanto, en condiciones de estado estacionario en un circuito CMOS no fluye corriente. Sin embargo, la corriente sí fluye en los circuitos CMOS durante corto tiempo, cuando las señales cambian de un nivel de voltaje a otro.

En la figura 3.50*a* se describe la situación siguiente. Supóngase que V_x ha estado en 0 V durante cierto tiempo; por ende, $V_f = 5$ V. Ahora, V_x cambia a 5 V. El transistor NMOS se enciende y lleva a V_f hacia *Gnd*. Debido a la capacitancia parásita *C* en el nodo *f*, el voltaje V_f no cambia de manera instantánea, y la corriente I_D fluye por el transistor NMOS durante breve tiempo mientras el capacitor se descarga. Una situación similar se presenta cuando V_x cambia de 5 V a 0, como se ilustra en la figura 3.50*b*. Aquí el capacitor *C* inicialmente tiene 0 V a través suyo y luego se carga a 5 V por el transistor PMOS. La corriente fluye de la fuente de poder a través del transistor PMOS mientras el capacitor se carga.

La característica de transferencia de voltaje del inversor CMOS mostrado en la figura 3.46 indica que hay un límite de voltaje de entrada V_x para el que ambos transistores en el inversor están encendidos. Dentro de ese límite, específicamente $V_T < V_x < (V_{DD} - V_T)$, la corriente fluye de V_{DD} a *Gnd* a través de ambos transistores. Esta corriente recibe el nombre de *corriente de cortocircuito* en la compuerta. Comparada con la cantidad de corriente usada para (des)cargar el capacitor *C*, la corriente de cortocircuito es despreciable en la mayoría de los casos.

La potencia que utiliza un solo inversor CMOS es extremadamente pequeña. Considérese de nuevo la situación planteada en la figura 3.50*a* cuando $V_f = V_{DD}$. La cantidad de energía almacenada en el capacitor es igual a $CV_{DD}^2/2$ (véase el ejemplo 3.12). Cuando el capacitor se descarga a 0 V, esta energía almacenada se disipa en el transistor NMOS. De manera similar, para la situación descrita en la figura 3.50*b*, la energía $CV_{DD}^2/2$ se disipa en el transistor PMOS cuando *C* se carga hasta V_{DD}. Por ende, para cada ciclo en el que el inversor carga y descarga *C*, la cantidad de energía disipada es igual a CV_{DD}^2. Puesto que la potencia se define como la energía utilizada por unidad de tiempo, la potencia disipada en el inversor es el producto de la energía usada en un ciclo de descarga/carga por el número de tales ciclos por segundo, *f*. En consecuencia,



a) Flujo de corriente cuando la entrada V_x cambia de 0 V a 5 V

b) Flujo de corriente cuando la entrada V_x cambia de 5 V a 0 V

Figura 3.50 Flujo de corriente dinámico en circuitos CMOS.

la potencia dinámica consumida es

$$P_D = fCV_{DD}^2$$

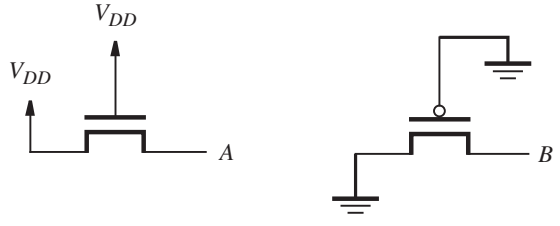
En la práctica, la cantidad total de potencia dinámica utilizada en los circuitos CMOS es significativamente más baja que la potencia total necesaria en otras tecnologías, como la NMOS. Por ello, prácticamente todos los grandes circuitos integrados que se fabrican en la actualidad se basan en tecnología CMOS.

Ejemplo 3.8 Para un inversor CMOS, suponga que $C = 70$ fF y $f = 100$ MHz. La potencia dinámica consumida por la compuerta es $P_D = 175$ μW. Si se asume que un chip contiene el equivalente de 10 000 inversores y que, en promedio, 20% de las compuertas cambia valores en un instante específico, entonces la cantidad total de potencia dinámica usada en el chip es $P_D = 0.2 \times 10\,000 \times 0.175$ μW $= 0.35$ mW.

3.8.7 PASO DE 1 Y 0 MEDIANTE INTERRUPTORES DE TRANSISTOR

En la figura 3.4 mostramos que los transistores NMOS se utilizan como dispositivos de bajada y los PMOS como dispositivos de subida. Ahora consideraremos el uso de los transistores en el sentido opuesto; es decir: de un transistor NMOS para dirigir una salida alta y de un transistor PMOS para una salida baja.

En la figura 3.51*a* se ilustra el caso de un transistor NMOS para el que tanto la terminal compuerta como un lado del interruptor se llevan a V_{DD}. Supóngase inicialmente que V_G y el nodo A están en 0 V, y luego V_G se cambia a 5 V. El nodo A es la terminal fuente del transistor porque tiene el voltaje más bajo. Como $V_{GS} = V_{DD}$, el transistor se enciende y dirige el nodo A hacia V_{DD}. Cuando el voltaje en el nodo A se eleva, V_{GS} disminuye hasta el punto en que V_{GS} ya no es mayor que V_T. En este punto el transistor se apaga. Por ello, en estado estacionario $V_A = V_{DD} - V_T$, lo que significa que un transistor NMOS sólo puede pasar parcialmente una señal de voltaje alto.



a) Transistor NMOS *b*) Transistor PMOS

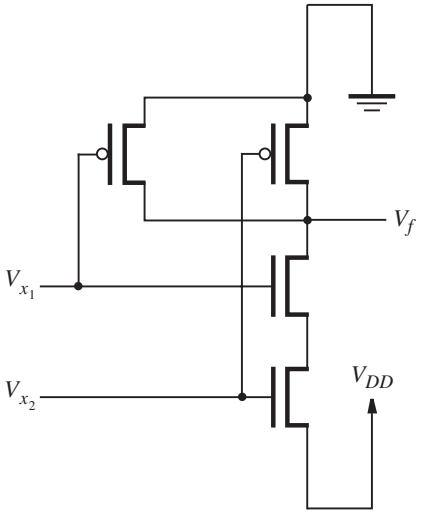
Figura 3.51 Transistores NMOS y PMOS empleados en la forma opuesta a la de la figura 3.4.

Una situación similar ocurre cuando se usa un transistor PMOS para pasar un nivel de voltaje bajo, como se muestra en la figura 3.51*b*. Aquí supóngase inicialmente que V_G y el nodo *B* están en 5 V. Luego V_G cambia a 0, de modo que el transistor se enciende y dirige el nodo fuente (nodo *B*) hacia 0 V. Cuando el nodo *B* disminuye a V_T, el transistor se apaga; por tanto, el voltaje de estado estacionario es igual a V_T.

En la sección 3.1 dijimos que para un transistor NMOS, la terminal sustrato (cuerpo) se conecta a *Gnd*, y para un transistor PMOS el sustrato se conecta a V_{DD}. El voltaje entre las terminales fuente y sustrato, V_{SB}, que se llama *voltaje de polarización del sustrato*, normalmente es igual a 0 V en un circuito lógico. Pero en la figura 3.51 ambos transistores, NMOS y PMOS, tienen $V_{SB} = V_{DD}$. El voltaje de polarización tiene el efecto de aumentar el voltaje umbral en el transistor V_T por un factor de aproximadamente 1.5 o más [2, 1]. Este conflicto se conoce como el *efecto cuerpo*.

Considérese la compuerta lógica mostrada en la figura 3.52. En este circuito, las conexiones V_{DD} y *Gnd* están invertidas de la forma en la que se usaron en los circuitos expuestos con anterioridad. Cuando V_{x_1} y V_{x_2} son altos, entonces V_f sube al voltaje de salida alto, $V_{OH} = V_{DD} - 1.5\ V_T$. Si $V_{DD} = 5$ V y $V_T = 1$ V, entonces $V_{OH} = 3.5$ V. Cuando V_{x_1} o V_{x_2} son bajos, entonces V_f baja hacia el voltaje de salida bajo, $V_{OL} = 1.5\ V_T$, o aproximadamente 1.5 V. Como se muestra en la tabla de verdad de la figura, el circuito representa una compuerta AND. En comparación con la compuerta AND normal que aparece en la figura 3.15, el circuito de la figura 3.52 parece mejor porque requiere menos transistores. Pero una desventaja de este circuito es que ofrece un margen de ruido menor debido a los pobres niveles de V_{OH} y V_{OL}.

Otra importante debilidad del circuito de la figura 3.52 es que provoca disipación de potencia estática, a diferencia de una compuerta AND CMOS normal. Supóngase que la salida de tal compuerta AND dirige la entrada de un inversor CMOS. Cuando $V_f = 3.5$ V, el transistor NMOS del inversor se enciende y el inversor de salida tiene un nivel de voltaje bajo. Pero el transistor



a) Circuito de compuerta AND

Valor lógico		Voltaje	Valor lógico
x_1	x_2	V_f	f
0	0	1.5 V	0
0	1	1.5 V	0
1	0	1.5 V	0
1	1	3.5 V	1

b) Tabla de verdad y niveles de voltaje

Figura 3.52 Implementación deficiente de una compuerta AND CMOS.

PMOS del inversor no se apaga, ya que su voltaje de compuerta a fuente es -1.5 V, que es mayor que V_T. A través del inversor fluye una corriente estática de V_{DD} a *Gnd*. Una situación similar ocurre cuando la compuerta AND produce la salida baja $V_f = 1.5$ V. Aquí, el transistor PMOS en el inversor se enciende, pero el transistor NMOS no se apaga. La implementación de la compuerta AND de la figura 3.52 no se usa en la práctica.

3.8.8 FACTORES DE CARGA DE ENTRADA Y DE SALIDA EN LAS COMPUERTAS LÓGICAS

El *factor de carga de entrada* (*fan-in*, también llamado *entrada en abanico*) de una compuerta lógica se define como el número de entradas a la compuerta. Según cómo se construya la compuerta lógica, puede ser impráctico aumentar el número de entradas más allá de una cantidad pequeña. Por ejemplo, considérese la compuerta NAND NMOS de la figura 3.53, que tiene k entradas. Queremos tener en cuenta el efecto de k en el retraso de propagación t_p a través de la compuerta. Supóngase que todos los k transistores NMOS tienen el mismo ancho W y longitud L. Puesto que los transistores están conectados en serie, es posible considerarlos como equiva-

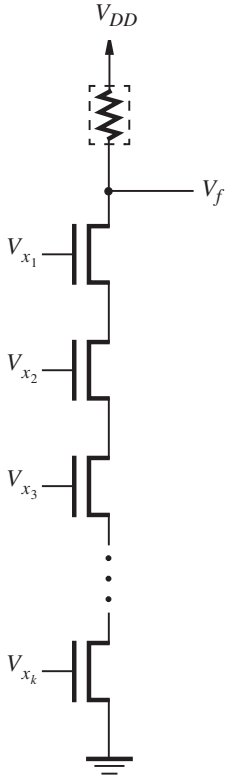


Figura 3.53 Compuerta NAND NMOS con factor de carga de entrada alto.

lentes a un largo transistor con longitud $k \times L$ y ancho W. Con la ecuación 3.4 (que puede aplicarse a ambas compuertas, CMOS y NMOS), el retraso de propagación está dado por

$$t_p \cong \frac{1.7\,C}{k_n' \frac{W}{L} V_{DD}} \times k$$

Aquí C es la capacitancia equivalente en la salida de la compuerta, incluida la capacitancia parásita aportada por cada uno de los k transistores. El rendimiento de la compuerta se puede mejorar un poco aumentando W para cada transistor NMOS. Pero este cambio ulteriormente aumenta C y es a expensas del área del chip. Otro inconveniente del circuito es que cada transistor NMOS tiene el efecto de aumentar V_{OL} y, por tanto, reducir el margen de ruido. Es práctico construir compuertas NAND de esta forma sólo si el factor de carga de entrada es pequeño.

Como otro ejemplo de factor de carga de entrada, en la figura 3.54 se muestra una compuerta NOR NMOS de k entradas. En este caso los k transistores NMOS conectados en paralelo pueden verse como un gran transistor con ancho $k \times W$ y longitud L. De acuerdo con la ecuación 3.4, el retraso de propagación debe disminuir por el factor k. Sin embargo, los transistores conectados en paralelo aumentan la capacitancia de carga C en la salida de la compuerta y, más importante, es en extremo improbable que todos ellos se enciendan cuando V_f cambia de un nivel alto a uno bajo. Por ende, es práctico construir compuertas NOR con factor de carga de entrada alto en tecnología NMOS. No obstante, cabe indicar que en una compuerta NMOS el retraso de propagación de bajo a alto puede ser más lento que el retraso de alto a bajo como resultado del efecto limitante de la corriente del dispositivo de subida (véanse los ejemplos 3.13 y 3.14).

Las compuertas lógicas CMOS con factor de carga de entrada alto siempre requieren k transistores NMOS o k PMOS en serie y, por consiguiente, nunca son prácticos. En CMOS la única forma razonable de construir una compuerta con factor de carga de entrada alto es usar dos o más compuertas con factor de carga de entrada más bajo. Por ejemplo, una forma de realizar una compuerta AND de seis entradas es como dos compuertas AND de tres entradas que conectan a una compuerta AND de dos entradas. Es posible construir una compuerta AND CMOS de seis entradas empleando menos transistores que los necesarios si se sigue este enfoque, lo que se deja como ejercicio para el lector (véase el problema 3.4).

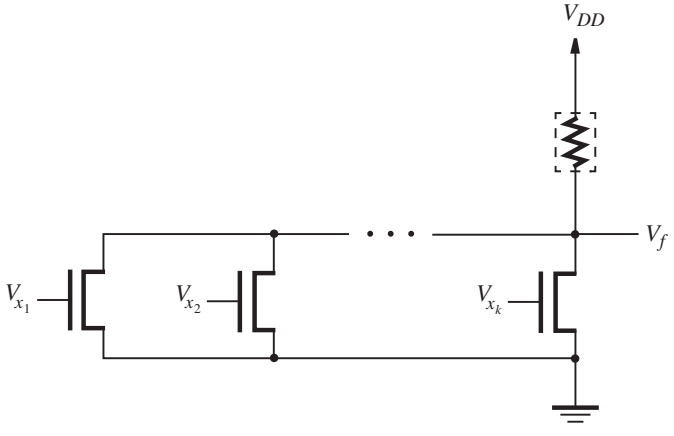
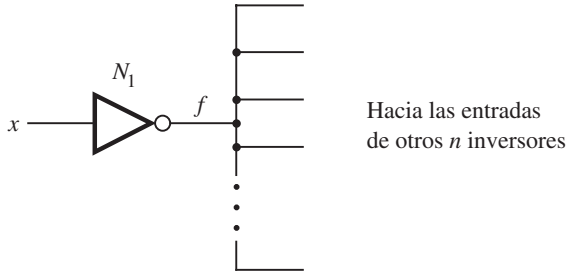


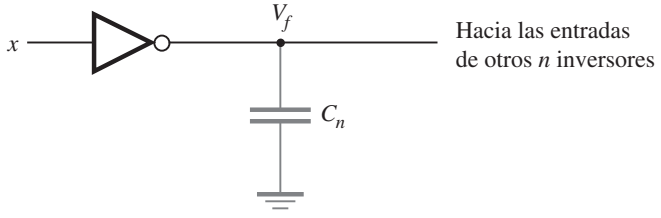
Figura 3.54 Compuerta NOR NMOS con factor de carga de entrada alto.

Factor de carga de salida

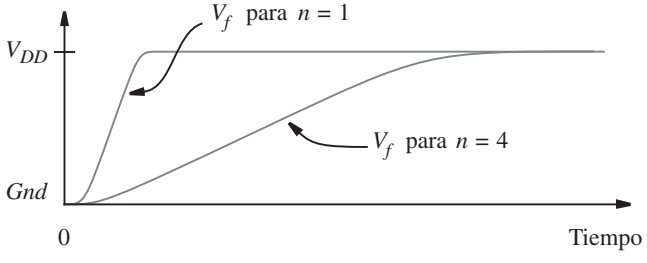
En la figura 3.48 se ilustraron retrasos de tiempo para una compuerta NOT que se dirige a otra. En los circuitos reales cada compuerta lógica puede ser necesaria para dirigir varias. El número de éstas que una compuerta específica dirige se denomina *factor de carga de salida* (*fan-out,* también llamado *salida en abanico*). Un ejemplo de factor de carga de salida se describe en la figura 3.55*a*, que muestra un inversor N_1 que dirige las entradas de otros n inversores. Cada uno de éstos contribuye a la carga capacitiva total en el nodo f. En el inciso *b*) de la figura, los n inversores se representan mediante un gran capacitor C_n. Por simplicidad, supóngase que cada inversor aporta una capacitancia C y que $C_n = n \times C$. La ecuación 3.4 muestra que el retraso de propagación aumenta en proporción directa a n.



a) Inversor que dirige otros *n* inversores



b) Circuito equivalente para propósitos de temporización



c) Tiempos de propagación para diferentes valores de *n*

Figura 3.55 El efecto del factor de carga de salida en el retraso de propagación.

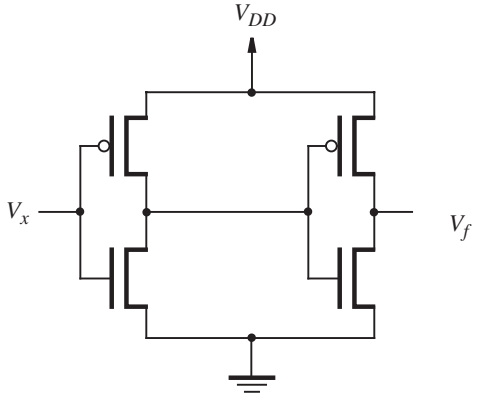
En la figura 3.55*c* se ilustra cómo *n* afecta el retraso de propagación. Supóngase que en el tiempo 0 ocurre un cambio en la señal *x*, del valor lógico 1 al 0. Una curva representa el caso donde $n = 1$, y la otra corresponde a $n = 4$. Si usamos los parámetros del ejemplo 3.7, cuando $n = 1$ se tiene $t_p = 0.1$ ns. Entonces, para $n = 4$, $t_p \approx 0.4$ ns. Es posible reducir t_p aumentando las razones W/L de los transistores en N_1.

Buffers

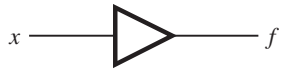
En los circuitos en los que una compuerta lógica debe dirigir una gran carga capacitiva se usan buffers para mejorar el rendimiento. Un *buffer* es una compuerta lógica con una entrada, *x*, y una salida, *f*, que produce $f = x$. La implementación más simple de un buffer usa dos inversores, como se muestra en la figura 3.56*a*. Los buffers pueden crearse con diferentes cantidades de *capacidad de dirección*, según el tamaño de los transistores (véase la figura 3.49). En general, puesto que se usan para dirigir cargas capacitivas más grandes que lo normal, los buffers tienen transistores que son más grandes que aquellos en las compuertas lógicas típicas. El símbolo gráfico para un buffer no inversor está dado en la figura 3.56*b*.

Otro tipo de buffer es el *buffer inversor*, que produce la misma salida que un inversor, $f = \overline{x}$, pero se construye con transistores relativamente grandes. El símbolo gráfico para el buffer inversor es el mismo que el de la compuerta NOT; un buffer inversor es justo una compuerta NOT capaz de dirigir grandes cargas capacitivas. En la figura 3.55, para grandes valores de *n* podría usarse un buffer inversor para el inversor etiquetado N_1.

Además de su uso para mejorar la velocidad de rendimiento de los circuitos, los buffers también sirven cuando se necesitan flujos de corriente altos para dirigir dispositivos externos. Los buffers pueden manejar cantidades de flujo de corriente hasta cierto punto grandes, ya



a) Implementación de un buffer



b) Símbolo gráfico

Figura 3.56 Un buffer no inversor.

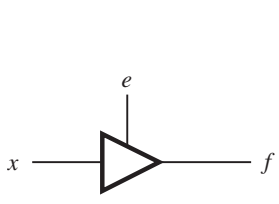
que se construyen con transistores grandes. Un ejemplo común de este uso de los buffers es el control de un diodo emisor de luz (LED). En la sección 7.14.3 se describe un ejemplo de esta aplicación.

En general, factor de carga de salida, carga capacitiva y flujo de corriente son cuestiones importantes que el diseñador de un circuito digital ha de considerar con sumo cuidado. En la práctica, la decisión de si se necesita buffers en un circuito o no se toma con la ayuda de herramientas CAD.

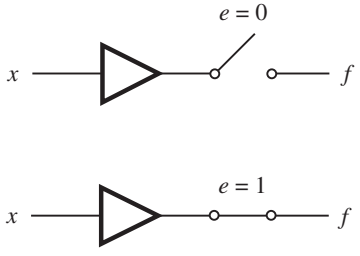
Buffers triestado

En la sección 3.6.2 mencionamos que un tipo de buffer, llamado *buffer triestado*, se incluye en algunos chips estándar y en PLD. Un buffer triestado tiene una entrada, x, una salida, f, y una entrada de control, llamada habilitador (*enable*), e. En la figura 3.57*a* se presenta el símbolo gráfico de un buffer triestado. La entrada habilitador sirve para determinar si el buffer triestado produce una señal de salida, como se ilustra en la figura 3.57*b*. Cuando $e = 0$, el buffer está completamente desconectado de la salida f; cuando $e = 1$, dirige el valor de x hacia f, lo que hace que $f = x$. Este comportamiento se describe en forma de tabla de verdad en el inciso *c*) de la figura. Para las dos filas de la tabla donde $e = 0$, la salida se denota mediante el valor lógico Z, que se llama *estado de alta impedancia*. El nombre *triestado* se deriva de que existen dos estados normales para una señal lógica, 0 y 1, y Z representa un tercer estado que no produce señal de salida. En la figura 3.57*d* se muestra una posible implementación del buffer triestado.

En la figura 3.58 se observan varios tipos de buffers triestado. El del inciso *b*) tiene el mismo comportamiento que el del inciso *a*), excepto que cuando $e = 1$, produce $f = \bar{x}$. En el inciso *c*) de la figura se presenta un buffer triestado para el que la señal habilitador tiene el comportamiento opuesto; esto es, cuando $e = 0, f = x$, y cuando $e = 1, f = Z$. Para describir este tipo de



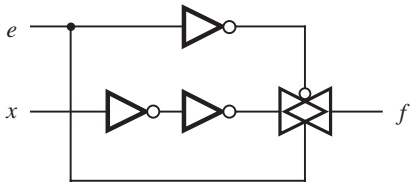
a) Buffer triestado



b) Circuito equivalente

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

c) Tabla de verdad



d) Implementación

Figura 3.57 Buffer triestado.

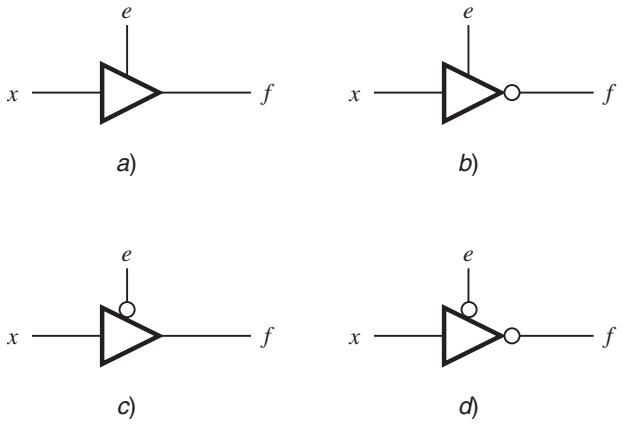


Figura 3.58 Cuatro tipos de buffers triestado.

comportamiento se dice que el habilitador está *activo bajo*. El buffer de la figura 3.58*d* también presenta un habilitador activo bajo y produce $f = \overline{x}$ cuando $e = 0$.

Como un pequeño ejemplo de cómo pueden usarse los buffers triestado considérese el circuito de la figura 3.59. En él la salida f es igual a x_1 o a x_2, según el valor de s. Cuando $s = 0, f = x_1$, y cuando $s = 1, f = x_2$. Los circuitos de este tipo, que eligen una de las entradas y reproducen la señal en esta entrada en la terminal de salida, se llaman circuitos *multiplexores*. En la figura 2.26 se muestra un circuito que implementa el multiplexor usando compuertas AND y OR. En la sección 3.9.2 presentaremos otra forma de construir circuitos multiplexores, y en el capítulo 6 los abordaremos en detalle.

En el circuito de la figura 3.59, las salidas de los buffers triestado están unidas por cables. Esta conexión es posible porque la entrada de control s está conectada de modo que se garantiza que uno de los dos buffers está en el estado de impedancia alta. El buffer x_1 está activo sólo cuando $s = 0$, y el x_2 sólo lo está cuando $s = 1$. Sería desastroso permitir que ambos estén activos al mismo tiempo. Hacerlo así crearía un cortocircuito entre V_{DD} y *Gnd* en cuanto los dos buffers produjeran diferentes valores. Por ejemplo, supóngase que $x_1 = 1$ y $x_2 = 0$. El buffer x_1 produce la salida V_{DD} y el x_2, *Gnd*. Entre V_{DD} y *Gnd* se forma un cortocircuito, a través de los transistores en los buffers triestado. La cantidad de corriente que fluye a través de tal cortocircuito basta para destruir el circuito.

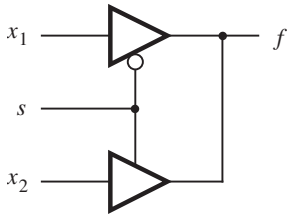


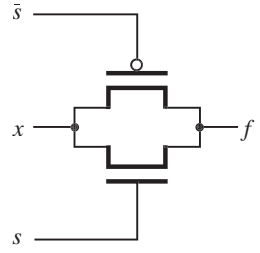
Figura 3.59 Una aplicación de los buffers triestado.

El tipo de conexión *cableada* que se usa para los buffers triestado no es posible con compuertas lógicas ordinarias, ya que sus salidas siempre están activas; por tanto, ocurriría un cortocircuito. Como ya se sabe, para circuitos lógicos normales, el resultado equivalente de la conexión cableada se logra empleando una compuerta OR para combinar señales, como se hace en la forma de suma de productos.

3.9 COMPUERTAS DE TRANSMISIÓN

En la sección 3.8.7 mostramos que un transistor NMOS pasa los 0 bien y los 1 mal, mientras que un transistor PMOS pasa los 1 bien y los 0 mal. Es posible combinar un transistor NMOS y uno PMOS en un solo interruptor que pueda dirigir su terminal de salida igualmente bien hacia un voltaje bajo o hacia uno alto. En la figura 3.60*a* se muestra el circuito de una *compuerta de transmisión*. Como se indica en los incisos *b*) y *c*) de la figura, funciona como un interruptor que conecta *x* a *f*. El control del interruptor lo proporciona la entrada de selección (*select*) *s* y su complemento $\overline{s}$. El interruptor se enciende al hacer $V_s = 5$ V y $V_{\overline{s}} = 0$. Cuando V_x es 0, el transistor NMOS se encenderá (porque $V_{GS} = V_s - V_x = 5$ V) y V_f será 0. Por otra parte, cuando V_x es 5 V, entonces el transistor PMOS estará en ($V_{GS} = V_{\overline{s}} - V_x = -5$ V) y V_f será 5 V. En la figura 3.60*d* se proporciona el símbolo gráfico de las compuertas de transmisión.

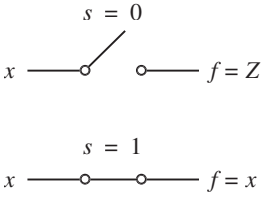
Las compuertas de transmisión pueden usarse en varias aplicaciones. A continuación demostraremos cómo conducen a implementaciones eficientes de las compuertas lógicas *OR exclusiva* (*XOR*) y los circuitos multiplexores.



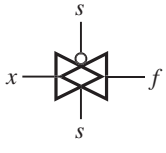
a) Circuito

s	*f*
0	Z
1	*x*

b) Tabla de verdad



c) Circuito equivalente



d) Símbolo gráfico

Figura 3.60 Una compuerta de transmisión.

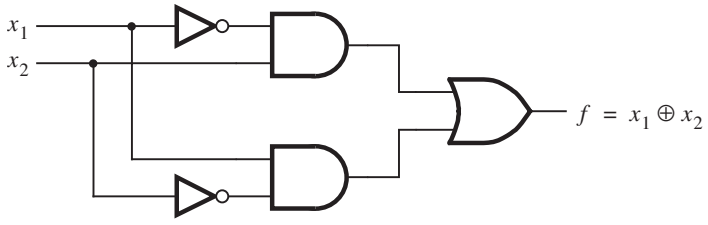
3.9.1 COMPUERTAS OR EXCLUSIVA

Hasta ahora hemos encontrado que las compuertas AND, OR, NOT, NAND y NOR son los elementos básicos a partir de los cuales se construyen los circuitos lógicos. Existe otro elemento básico muy útil en la práctica, sobre todo para crear circuitos que realizan operaciones aritméticas, como veremos en el capítulo 5. Este elemento cumple la función OR exclusiva que se define en la figura 3.61*a*. La tabla de verdad de esta función es similar a la función OR excepto que $f = 0$ cuando ambas entradas son 1. Por tal similitud, la función se llama OR exclusiva, que suele abreviarse XOR. El símbolo gráfico de una compuerta que implementa XOR se presenta en el inciso *b*) de la figura.

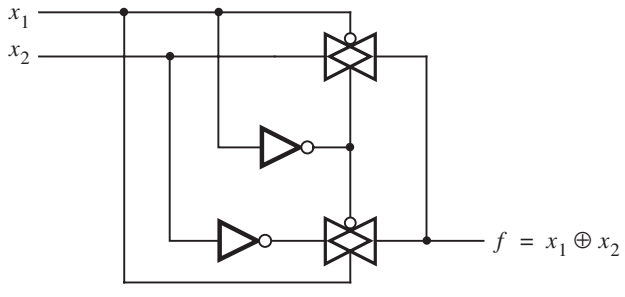
x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

a) Tabla de verdad

b) Símbolo gráfico



c) Implementación en suma de productos



d) Implementación CMOS

Figura 3.61 Compuerta OR exclusiva.

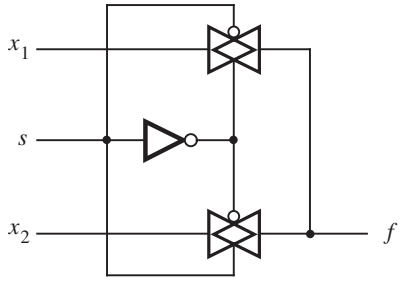


Figura 3.62 Un multiplexor 2 a 1 construido con compuertas de transmisión.

Usualmente, la operación XOR se denota con el símbolo $\oplus$. Puede realizarse en la forma de suma de productos como

$$x_1 \oplus x_2 = \overline{x}_1 x_2 + x_1 \overline{x}_2$$

que conduce al circuito de la figura 3.61*c*. En la sección 3.3 explicamos que cada compuerta AND y OR requiere seis transistores, mientras que una compuerta NOT necesita dos. Por tanto, se precisan 22 transistores para implementar este circuito en tecnología CMOS. Es posible reducir mucho el número de transistores necesarios si se emplean compuertas de transmisión. En la figura 3.61*d* se proporciona un circuito para una compuerta XOR que utiliza dos compuertas de transmisión y dos inversores. La salida f se establece en el valor de x_2 cuando $x_1 = 0$ por la compuerta de transmisión superior. La compuerta de transmisión inferior establece f en $\overline{x}_2$ cuando $x_1 = 1$. El lector puede comprobar que este circuito implementa bien la función XOR. En el capítulo 6 mostraremos cómo se derivan tales circuitos.

3.9.2 CIRCUITO MULTIPLEXOR

En la figura 3.59 mostramos cómo construir un multiplexor con buffers triestado. Se puede usar una estructura semejante para realizar un multiplexor con compuertas de transmisión, como se indica en la figura 3.62. La entrada de selección s se usa para elegir si la salida f debe tener el valor de entrada x_1 o x_2. Si $s = 0$, entonces $f = x_1$; si $s = 1$, entonces $f = x_2$.

3.10 DETALLES DE IMPLEMENTACIÓN PARA SPLD, CPLD Y FPGA

En la sección 3.6 presentamos los PLD. En los diagramas de chip mostrados allí los interruptores programables se representaron con el símbolo X. Ahora veremos la manera de implementar tales interruptores mediante transistores.

En los SPLD comerciales se usan dos tecnologías principales para fabricar los interruptores programables. La tecnología más antigua se basa en el uso de fusibles de aleación metálica como vínculos programables. En esta tecnología los PLA y las PAL se fabrican de modo que cada

par de cables horizontales y verticales que se cruzan se conectan mediante un pequeño fusible metálico. Cuando el chip se programa, por cada conexión que no se quiera implementar en el circuito el fusible asociado se derrite. El proceso de programación no es reversible, ya que los fusibles derretidos se destruyen. No ahondaremos en esta tecnología, pues casi se ha sustituido totalmente por un método más nuevo y mejor.

En los PLA y las PAL que se producen hoy día los interruptores programables se implementan usando un tipo especial de *transistores programables*. Como los CPLD comprenden bloques parecidos a PAL, la tecnología empleada en los SPLD también es aplicable a los CPLD. Ilustraremos las ideas principales describiendo primero los PLA. Para que un PLA sea útil en la implementación de un amplio repertorio de funciones lógicas debe soportar funciones de unas cuantas variables y funciones de muchas variables. En la sección 3.8.8 abordamos el tema del factor de carga de entrada en las compuertas lógicas. Mostramos que, cuando el factor de carga de entrada es alto, el mejor tipo de compuerta por usar es la NOR NMOS. Por tanto, los PLA usualmente se basan en este tipo de compuerta.

Como pequeño ejemplo de implementación PLA, considérese el circuito de la figura 3.63. El cable horizontal etiquetado con S_1 es la salida de una compuerta NOR NMOS con las entradas x_2 y $\bar{x}_3$. Por ende, $S_1 = \overline{x_2 + \bar{x}_3}$. De manera similar, S_2 y S_3 son las salidas de compuertas NOR

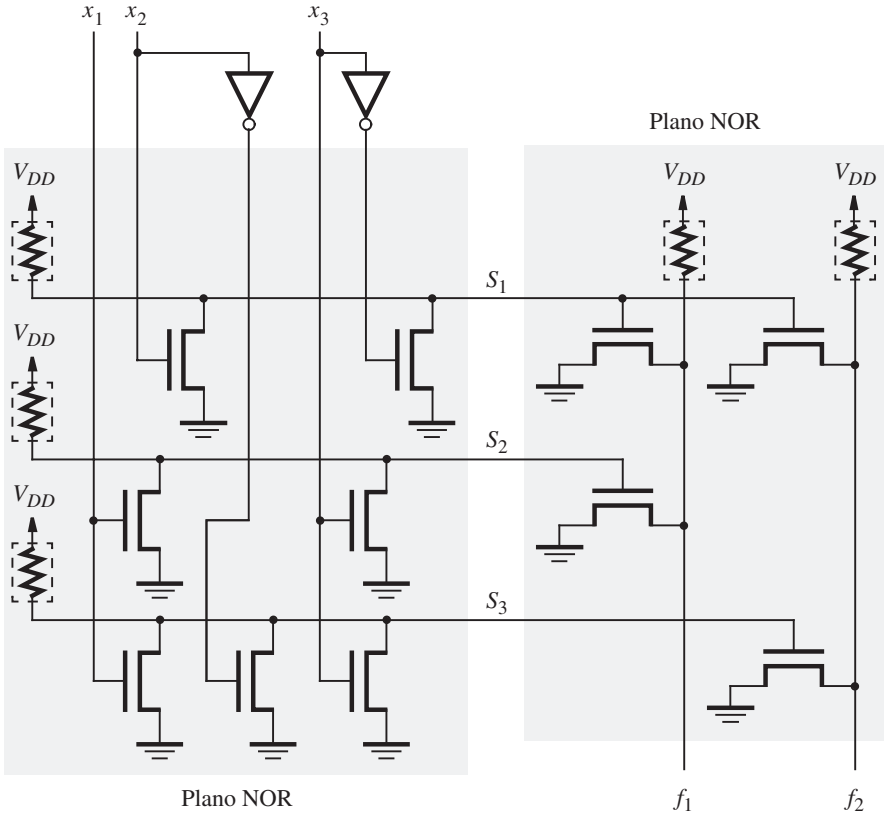


Figura 3.63 Ejemplo de un PLA NOR-NOR.

que producen $S_2 = \overline{x_1 + x_3}$ y $S_3 = \overline{x_1 + \overline{x}_2 + x_3}$. Las tres compuertas NOR que producen S_1, S_2 y S_3 se ordenan en una estructura regular que es eficiente para crear un circuito integrado. Esta estructura se llama *plano NOR*. El plano NOR se extiende a tamaños más grandes añadiendo columnas para entradas adicionales y filas para más compuertas NOR.

Las señales S_1, S_2 y S_3 sirven como entradas a un segundo plano NOR, el cual se gira 90 grados en el sentido de las manecillas del reloj respecto al primer plano NOR para que el diagrama sea más fácil de trazar. La compuerta NOR que produce la salida f_1 tiene las entradas S_1 y S_2. Por tanto

$$f_1 = \overline{S_1 + S_2} = \overline{(\overline{x_2 + \overline{x}_3}) + (\overline{x_1 + x_3})}$$

Si aplicamos el teorema de DeMorgan, esta expresión equivale a la expresión en producto de sumas

$$f_1 = \overline{S}_1\overline{S}_2 = (x_2 + \overline{x}_3)(x_1 + x_3)$$

De manera similar, la compuerta NOR con salida f_2 tiene entradas S_1 y S_3. En consecuencia,

$$f_2 = \overline{S_1 + S_3} = \overline{(\overline{x_2 + \overline{x}_3}) + (\overline{x_1 + \overline{x}_2 + x_3})}$$

que equivale a

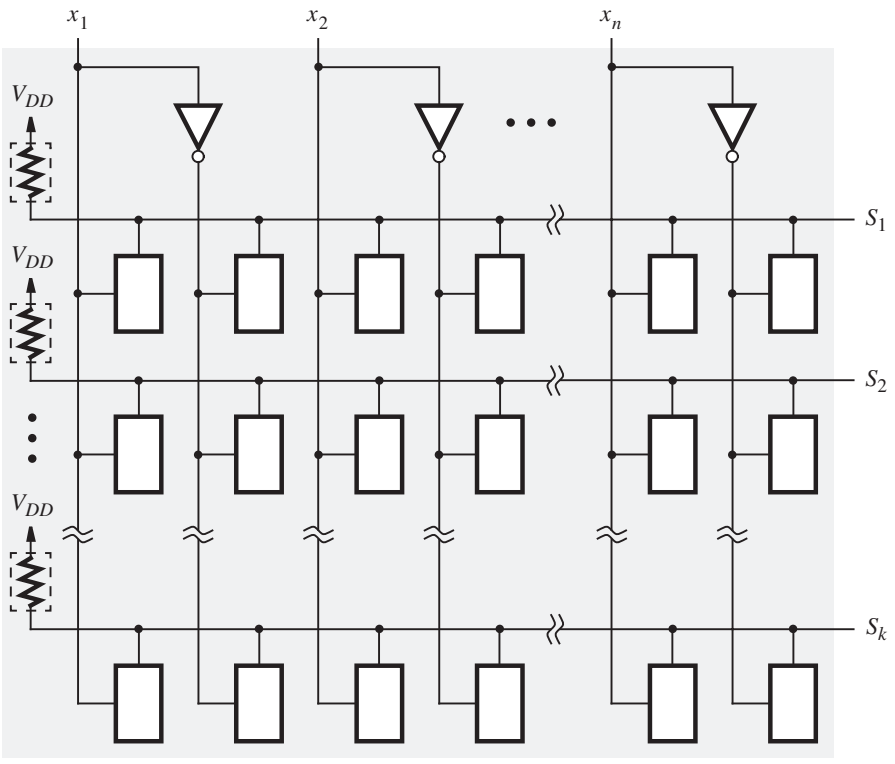
$$f_2 = \overline{S}_1\overline{S}_3 = (x_2 + \overline{x}_3)(x_1 + \overline{x}_2 + x_3)$$

El estilo de PLA ilustrado en la figura 3.63 se llama *PLA NOR-NOR*. Aunque también hay otras implementaciones, el estilo NOR-NOR es el más popular debido a su simplicidad. El lector debe notar que el PLA de la figura 3.63 no es programable: con los transistores conectados como se muestra sólo realiza las dos funciones lógicas específicas f_1 y f_2. Pero la estructura NOR-NOR puede utilizarse en una versión programable del PLA, como explicamos a continuación.

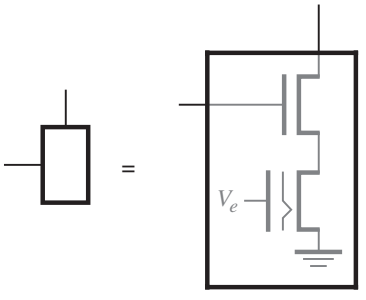
Estrictamente hablando, el término PLA debe usarse sólo para el tipo fijo de PLA descrito en la figura 3.63. El término técnico apropiado para un tipo programable de PLA es *arreglo lógico de campos programables* (FPLA, *field-programmable logic array*). Sin embargo, es uso común omitir la *F*. En la figura 3.64*a* se muestra una versión programable de un plano NOR. Tiene n entradas, $x_1, \ldots, x_n$, y k salidas, $S_1, \ldots, S_k$. En cada punto de cruce de un cable horizontal y otro vertical existe un interruptor programable. Éste comprende dos transistores conectados en serie, uno NMOS y uno de *memoria programable de sólo lectura eléctricamente borrable* (EEPROM, *electrically erasable programmable read-only memory*).

El interruptor programable se basa en el comportamiento del transistor EEPROM. Los libros de electrónica, como [1, 2], brindan explicaciones detalladas de cómo operan los transistores EEPROM. Aquí sólo daremos una breve descripción. En la figura 3.64*b* se presenta un interruptor programable y en la 3.64*c*, la estructura del transistor EEPROM. Éste muestra la misma apariencia general que un transistor NMOS (véase la figura 3.43) con una gran diferencia. El transistor EEPROM posee dos compuertas: la compuerta normal que tiene un transistor NMOS y una segunda compuerta flotante. La *compuerta flotante* se llama así porque está rodeada por vidrio aislador y no se conecta a parte alguna del transistor. Cuando este último se halla en el estado original sin programar, la compuerta flotante no tiene efecto en la operación del transistor y funciona como un transistor NMOS normal. Durante el uso normal del PLA, el voltaje en la compuerta flotante V_e se establece en V_{DD} mediante circuitos no mostrados en la figura, y el transistor EEPROM se enciende.

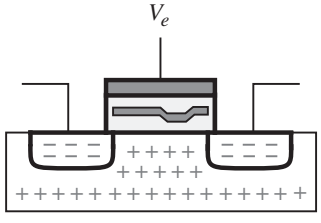
La programación del transistor EEPROM se logra encendiendo el transistor con un nivel de voltaje más alto que lo normal (por lo general, $V_e = 12$ V), que ocasiona que una gran cantidad



a) Plano NOR programable



b) Interruptor programable



c) Transistor EEPROM

Figura 3.64 Uso de transistores EEPROM para crear un plano NOR programable.

de corriente fluya por el canal del transistor. En la figura 3.64*c* se muestra que una parte de la compuerta flotante se extiende hacia abajo de modo que está muy cerca de la superficie superior del canal. Una corriente alta que fluye a través del canal genera un efecto, conocido como túnel *Fowler-Nordheim*, en el que algunos de los electrones en el canal "forman un túnel" a través del vidrio aislador en su punto más delgado y quedan atrapados bajo la compuerta flotante. Después

de completar el proceso de programación, los electrones atrapados repelen a otros para evitar que entren en el canal. Cuando el voltaje $V_e = 5$ V se aplica al transistor EEPROM, que normalmente causaría su encendido, los electrones atrapados mantienen al transistor apagado. En consecuencia, en el plano NOR de la figura 3.64*a*, la programación se usa para "desconectar" entradas de las compuertas NOR. Para las entradas que deben conectarse a cada compuerta NOR, los correspondientes transistores EEPROM se dejan en el estado no programado.

Una vez que un transistor EEPROM se programa, retiene permanentemente su estado. Sin embargo, el proceso de programación puede revertirse. Este paso se llama *borrado* y se lleva a cabo usando voltajes de polaridad opuesta a los ocupados para la programación. En este caso, el voltaje aplicado hace que los electrones atrapados bajo la compuerta flotante regresen a través del túnel. El transistor EEPROM regresa a su estado original y de nuevo funciona como un transistor NMOS normal.

Para decirlo todo, cabe también mencionar otra tecnología similar a la EEPROM, llamada *PROM borrable* (*EPROM*). Este tipo de transistor, que en realidad se creó como predecesor del EEPROM, se programa de forma semejante a la del EEPROM. No obstante, el borrado se hace de manera diferente: para borrar un transistor EPROM hay que exponerlo a energía luminosa de longitudes de onda específicas. Para facilitar este proceso, los chips basados en tecnología EPROM se albergan en paquetes con una ventana de vidrio transparente a través de la cual puede verse el chip. Para borrar el contenido de un chip se lo coloca bajo una fuente de *luz ultravioleta* durante varios minutos. Puesto que borrar transistores EPROM es más complicado que el proceso eléctrico utilizado para borrar transistores EEPROM, esencialmente la tecnología EPROM se sustituyó en la práctica con la tecnología EEPROM.

En la figura 3.65 se describe un PLA NOR-NOR completo que utiliza tecnología EEPROM, tiene cuatro entradas, seis términos suma en el primer plano NOR y dos salidas. Cada interruptor programable que se programa al estado apagado (*off*) se muestra como una X en negro, y cada interruptor que se queda sin programar se muestra en gris. Con los estados de programación indicados en la figura, el PLA realiza las funciones lógicas $f_1 = (x_1 + x_3)(x_1 + \overline{x}_2)(\overline{x}_1 + x_2 + \overline{x}_3)$ y $f_2 = (x_1 + \overline{x}_3)(\overline{x}_1 + x_2)(x_1 + \overline{x}_2)$.

Más que implementar funciones lógicas en forma de producto de sumas, un PLA también sirve para realizar la forma de suma de productos. Para ésta es preciso implementar compuertas AND en el primer plano NOR del PLA. Si primero se complementan las entradas a dicho plano, entonces, de acuerdo con el teorema de DeMorgan, esto es equivalente a la creación de un plano AND. En el PLA pueden generarse los complementos sin costo alguno, pues cada entrada ya se ofrece tanto en forma verdadera como complementada. Un ejemplo que ilustra la implementación de la forma en suma de productos se presenta en la figura 3.66. Las salidas del primer plano NOR están etiquetadas con $P_1, \ldots, P_6$ para reflejar nuestra interpretación de ellas como términos producto. La señal P_1 se programa para realizar $\overline{\overline{x}_1 + \overline{x}_2} = x_1x_2$. De manera similar, $P_2 = x_1\overline{x}_3$, $P_3 = \overline{x}_1\overline{x}_2x_3$, y $P_4 = \overline{x}_1\overline{x}_2\overline{x}_3$. Tras generar los términos producto deseados, ahora se necesita aplicarles la operación OR. Ello se logra mediante la complementación de las salidas del segundo plano NOR. En la figura 3.66 se han incluido compuertas NOT para tal propósito. Los estados indicados para los interruptores programables en el plano OR (el segundo plano NOR) de la figura producen las salidas siguientes: $f_1 = P_1 + P_2 + P_3 = x_1x_2 + x_1\overline{x}_3 + \overline{x}_1\overline{x}_2x_3$, y $f_2 = P_1 + P_4 = x_1x_2 + \overline{x}_1\overline{x}_2\overline{x}_3$.

Los conceptos antes descritos para los PLA también son aplicables a las PAL. En la figura 3.67 se muestra una PAL con cuatro entradas y dos salidas. Supóngase que el primer plano NOR se programa para realizar los términos producto en la forma descrita líneas arriba. Obsérvese en

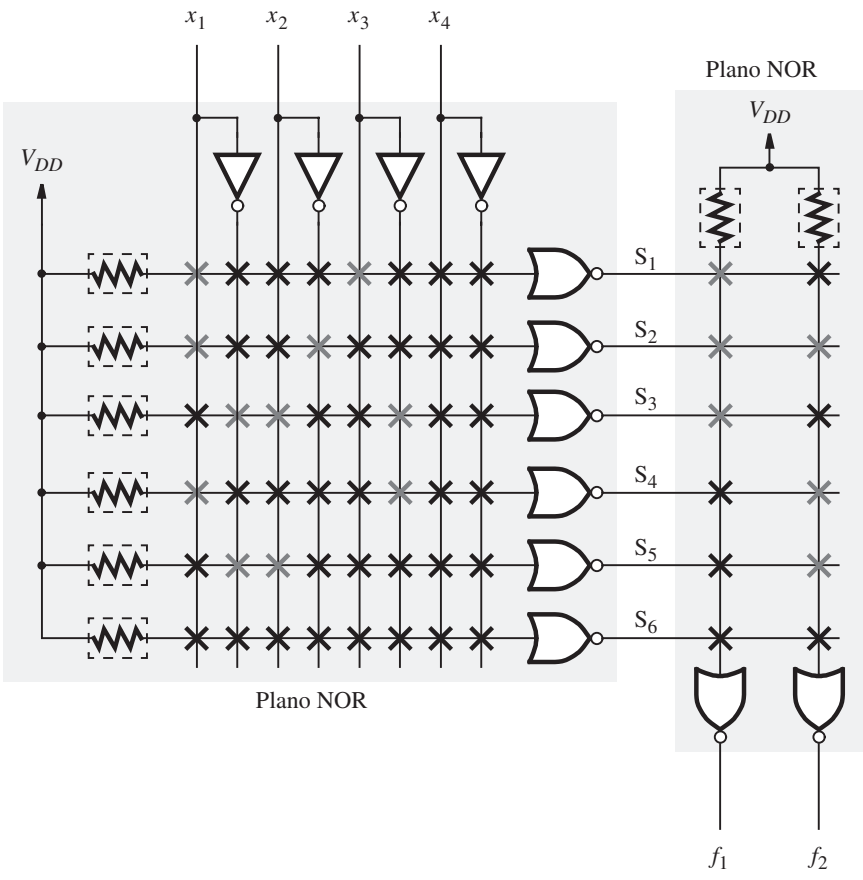


Figura 3.65 Versión programable del PLA NOR-NOR.

la figura que los términos producto están unidos con cable en grupos de tres a compuertas OR que producen las salidas de la PAL. Como ilustramos en la figura 3.29, la PAL también puede contener circuitos adicionales entre las compuertas OR y los pines de salida, que no se muestran en la figura 3.67. La PAL se programa para cumplir las mismas funciones lógicas, f_1 y f_2, que se generaron en el PLA de la figura 3.66. Obsérvese que el término producto x_1x_2 se implementó dos veces en el PAL, tanto en P_1 como en P_4. La duplicación es necesaria porque en una PAL los términos producto no pueden compartirse mediante salidas múltiples, lo que sí puede hacerse en un PLA. Otro detalle que cabe advertir en la figura 3.67 es que si bien la función f_2 requiere sólo dos términos producto, cada compuerta OR está unida por cable a tres términos producto. El término producto adicional P_6 debe establecerse en el valor lógico 0, de modo que no tenga efecto. Esto se logra programando P_6 para que produzca el producto de una entrada y el complemento de ésta, que siempre resulta en 0. En la figura, $P_6 = x_1\bar{x}_1 = 0$, pero para este propósito también podría emplearse cualquier otra entrada.

Los bloques parecidos a PAL contenidos en los CPLD casi siempre se implementan mediante la aplicación de las técnicas explicadas en esta sección. En un CPLD típico, el plano AND se construye con compuertas NOR NMOS, con la adecuada complementación de las entradas.

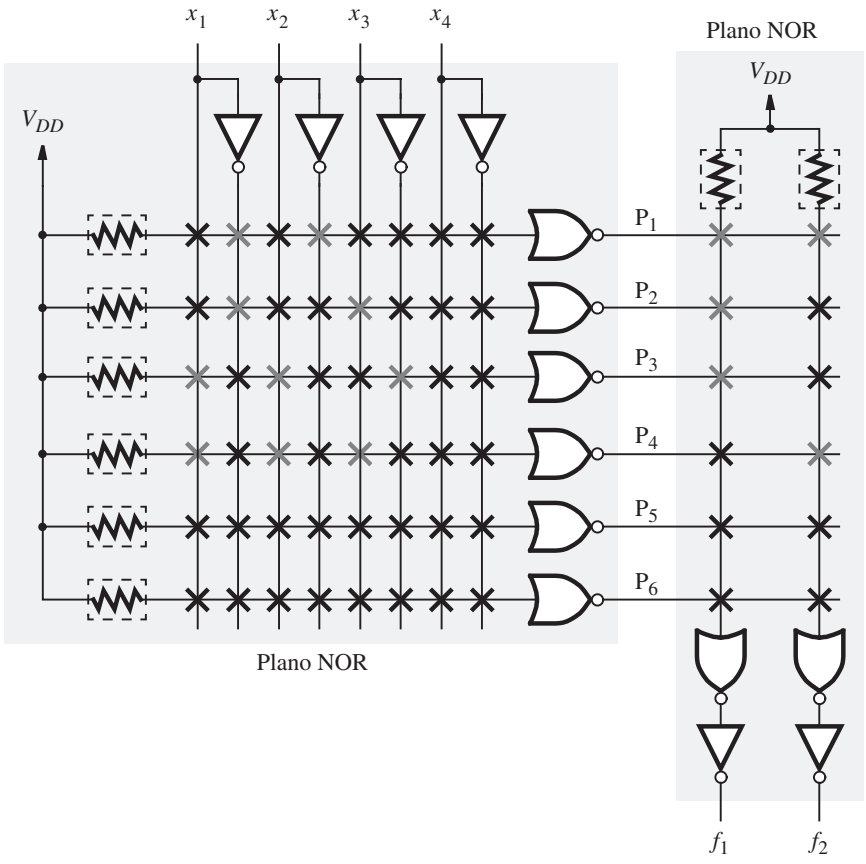


Figura 3.66 PLA NOR-NOR utilizado para suma de productos.

El plano OR tiene los cables igual que en una PAL, en vez de ser completamente programable como en un PLA. Sin embargo, hay cierta flexibilidad en el número de términos producto que alimentan cada compuerta OR. Esta flexibilidad se logra mediante un circuito programable que puede asignar los términos producto a cualquiera de las compuertas OR que el usuario desee. Un ejemplo de este tipo de flexibilidad, que ofrece un CPLD comercial, se proporciona en el apéndice E.

3.10.1 IMPLEMENTACIÓN EN FPGA

Los FPGA no usan tecnología EEPROM para implementar los interruptores programables. En vez de ello, la información de programación se almacena en celdas de memoria, llamadas *celdas de memoria estática de acceso aleatorio* (SRAM, *static random access memory*). La operación de este tipo de celda de almacenamiento se describe con hondura en la sección 10.1.3. Por ahora baste saber que cada una puede almacenar un 0 o un 1 lógicos, y proporciona este valor almacenado como salida. Por cada valor de la tabla de verdad almacenado en una LUT se usa una

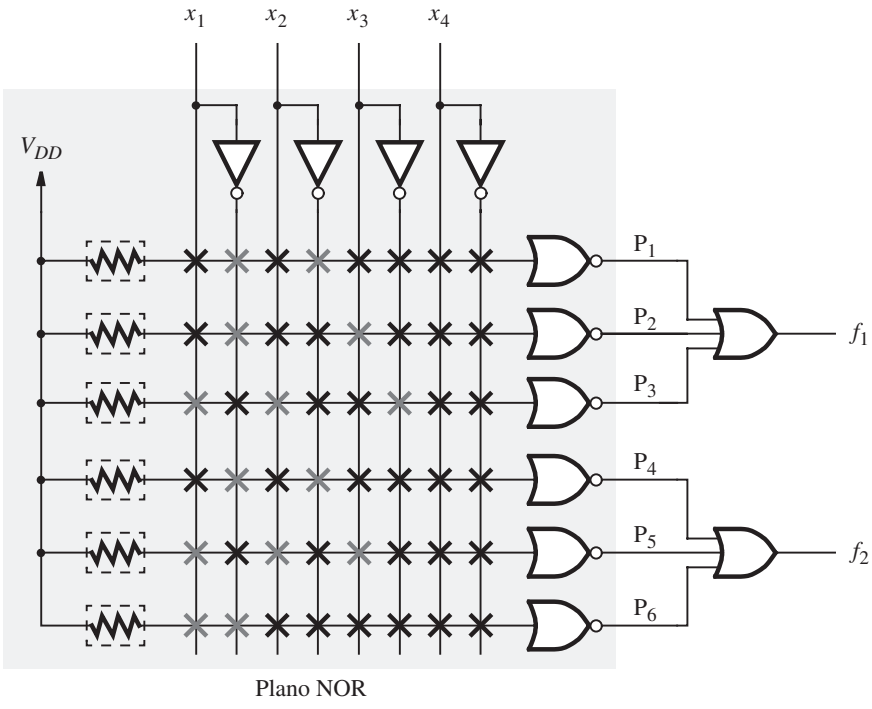


Figura 3.67 PAL programada para implementar las funciones de la figura 3.66.

celda SRAM. Este tipo de celdas también sirven para configurar los cables de interconexión en un FPGA.

En la figura 3.68 se describe una pequeña sección del FPGA de la figura 3.39. El bloque lógico mostrado produce la salida f_1, que se dirige hacia el cable horizontal mostrado en gris. Este cable puede conectarse a alguno de los alambres verticales que cruza por medio de inte-

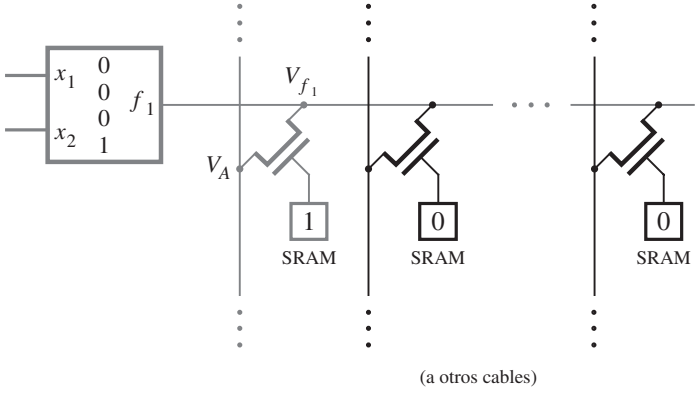


Figura 3.68 Interruptores de transistores de paso en FPGA.

rruptores programables. Cada interruptor se implementa con un transistor NMOS, con su terminal compuerta controlada por una celda SRAM. Tal interruptor se conoce como *interruptor de transistor de paso*. Si un 0 se almacena en una celda SRAM, entonces el transistor NMOS asociado se apaga. Pero si se almacena un 1, como se muestra para el interruptor dibujado en gris, ese transistor se enciende. Este interruptor forma una conexión entre los dos cables unidos a sus terminales fuente y drenado. El número de interruptores contenidos en el FPGA depende de la arquitectura de chip. En ciertos FPGA algunos de los interruptores se implementan con buffers triestado, en vez de transistores de paso. En el apéndice E se presentan ejemplos de chips FPGA comerciales.

En la sección 3.8.7 demostramos que un transistor NMOS sólo puede pasar parcialmente un valor lógico alto; por tanto, en la figura 3.68, si V_{f1} es un nivel de voltaje alto, entonces V_A sólo es parcialmente alto. Al usar los valores de la sección 3.8.7, si $V_{f1} = 5$ V, entonces $V_A = 3.5$ V. Como explicamos en la sección 3.8.7, este nivel de voltaje degradado hace que se consuma potencia estática (véase el ejemplo 3.15). Una solución a este problema [1] se ilustra en la figura 3.69. Suponemos que la señal V_A pasa a través de otro interruptor de transistor de paso antes de alcanzar su destino en otro bloque lógico. La señal V_B tiene el mismo valor que V_A porque la caída del voltaje umbral sólo ocurre cuando pasa por el primer interruptor de transistor de paso. Para restaurar el nivel de V_B se protege con un inversor. Entre la entrada del inversor y V_{DD} se conecta un transistor PMOS, que se controla mediante la salida del inversor. El transistor PMOS no tiene efecto en el nivel de voltaje de salida del inversor cuando $V_B = 0$ V. Pero cuando $V_B = 3.5$ V, entonces la salida del inversor es baja, lo que enciende el transistor PMOS. Éste rápidamente restaura V_B al nivel adecuado de V_{DD}, con lo que evita que fluya corriente en el estado estacionario. En lugar de utilizar esta solución de transistor de subida, otro posible enfoque es alterar el voltaje umbral del transistor PMOS (durante el proceso de fabricación del circuito integrado) en el inversor de la figura 3.69, tal que su magnitud sea lo suficientemente grande como para evitar que el transistor se apague cuando $V_B = 3.5$ V. En los FPGA comerciales se usan ambas soluciones en diferentes chips.

Una alternativa al uso de un solo transistor NMOS consiste en emplear una compuerta de transmisión, descrita en la sección 3.9, por cada interruptor. Si bien esto resuelve el problema del nivel de voltaje, plantea dos inconvenientes. Primero, tener un transistor NMOS y otro PMOS en

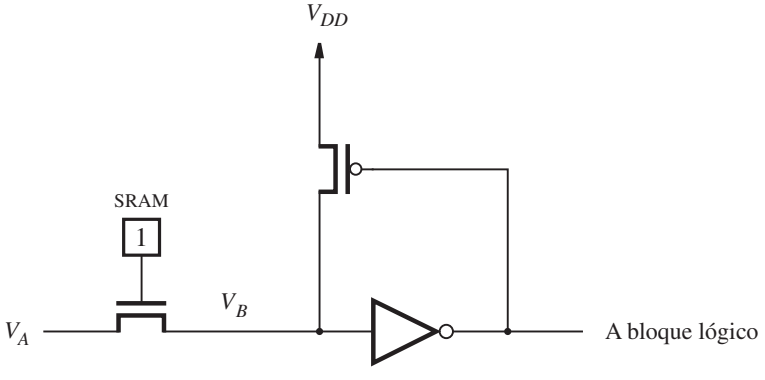


Figura 3.69 Restauración de un nivel de voltaje alto.

el interruptor incrementa la carga capacitiva en los cables de interconexión, lo que aumenta los retrasos de propagación y el consumo de potencia. Segundo, la compuerta de transmisión ocupa más área de chip que un solo transistor NMOS. Por estas razones, en la actualidad los chips FPGA comerciales no utilizan interruptores de compuerta de transmisión.

3.11 COMENTARIOS FINALES

Hemos descrito los conceptos más importantes necesarios para entender cómo se construyen las compuertas lógicas usando transistores. Nuestra explicación de la fabricación de transistores, los niveles de voltaje, los retrasos de propagación, la disipación de potencia y aspectos similares tiene la intención de dar al lector una apreciación de los conflictos prácticos que han de considerarse al diseñar y utilizar circuitos lógicos.

Presentamos varios tipos de chips de circuitos integrados. Cada uno es apropiado para tipos específicos de aplicaciones. Los chips estándar, como la serie 7400, contienen sólo unas cuantas compuertas simples y hoy día se usan rara vez. Las excepciones a esto son los chips buffer, que se emplean en circuitos digitales que deben dirigir grandes cargas capacitivas a altas velocidades. Los diversos tipos de PLD se usan mucho en diversos tipos de aplicaciones. Los PLD simples, como los PLA y las PAL, son adecuados para implementar pequeños circuitos lógicos. Los SPLD ofrecen bajo costo y alta velocidad. Los CPLD sirven para las mismas aplicaciones que los SPLD, pero también para la implementación de circuitos grandes, hasta de aproximadamente 10 000 o 20 000 compuertas. Muchas de las aplicaciones en las que pueden emplearse los CPLD también pueden realizarse con FPGA. Cuál de estos dos tipos de chips se usará en una situación de diseño específica depende de muchos factores. Si se sigue la tendencia de colocar cuantos circuitos sea posible en un solo chip, los CPLD y los FPGA son mucho más útiles que los SPLD. El grueso de los diseños digitales creados en la industria de hoy contiene algún tipo de PLD.

Las tecnologías de arreglo de compuertas, celda estándar y chip a la medida se utilizan en casos en que los PLD no son apropiados. Las aplicaciones típicas son aquellas que suponen circuitos muy grandes, que requieren velocidad de operación muy elevada, necesitan bajo consumo de potencia y donde se espera que los productos diseñados se vendan en grandes cantidades.

En el capítulo siguiente se examina el problema de la optimación de las funciones lógicas. Algunas de las técnicas recién expuestas son apropiadas para la síntesis de circuitos lógicos sin importar qué tipo de tecnología se use para la implementación. Otras técnicas son adecuadas para sintetizar circuitos de modo que puedan implementarse en chips con tipos específicos de recursos. Demostraremos que cuando se sintetiza una función lógica para crear un circuito, los métodos de optimación usados dependen, al menos en parte, del tipo de chip usado.

3.12 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas típicos que el lector puede encontrar, al tiempo que se muestra cómo pueden resolverse.

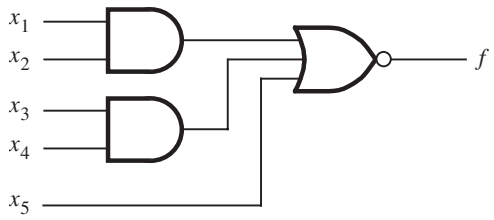


Figura 3.70 La celda AOI del ejemplo 3.9.

Ejemplo 3.9 **Problema:** En la sección 3.7 expusimos la tecnología de celda estándar, en la que los circuitos se crean mediante la interconexión de celdas de bloques constructores que implementan funciones simples, como compuertas lógicas básicas. Un tipo de celda estándar usado comúnmente son las celdas *and-or-invert* (AOI), que pueden fabricarse de manera eficiente como compuertas complejas CMOS. Considérese la celda AOI mostrada en la figura 3.70. Esta celda implementa la función $f = \overline{x_1x_2 + x_3x_4 + x_5}$. Derive la compuerta compleja CMOS que implementa esta celda.

Solución: Al aplicar el teorema de DeMorgan en dos pasos se obtiene

$$\begin{aligned} f &= \overline{x_1x_2} \cdot \overline{x_3x_4} \cdot \overline{x_5} \\ &= (\overline{x}_1 + \overline{x}_2) \cdot (\overline{x}_3 + \overline{x}_4) \cdot \overline{x}_5 \end{aligned}$$

Puesto que todas las variables de entrada se complementan en esta expresión, podemos derivar de modo directo la red de subida que tiene transistores PMOS conectados en paralelo y controlados por x_1 y x_2, en serie con transistores conectados en paralelo controlados por x_3 y x_4, en serie con un transistor controlado por x_5. Este circuito, junto con la correspondiente red de bajada, se muestra en la figura 3.71.

Ejemplo 3.10 **Problema:** Para la compuerta compleja CMOS de la figura 3.71, determine el tamaño de los transistores que deben usarse a fin de que la velocidad de rendimiento de esta compuerta sea similar a la de un inversor.

Solución: Recuerde de la sección 3.8.5 que un transistor con longitud L y ancho W tiene una fuerza de dirección proporcional a la razón W/L. También recuerde que cuando los transistores se conectan en paralelo, sus anchos se suman efectivamente, lo que desemboca en un aumento en la fuerza de dirección. Asimismo, cuando los transistores se conectan en serie sus longitudes se suman, lo que conduce a una disminución en la fuerza de dirección. Suponga que todos los transistores NMOS y PMOS tienen la misma longitud, $L_n = L_p = L$. Para la red de bajada de la figura 3.71, la trayectoria de peor caso comprende justo un solo transistor NMOS. Por ende, se puede hacer que la longitud, L_n, de cada transistor NMOS tenga el mismo tamaño que el inversor. Para la red de subida, la trayectoria de peor caso abarca tres transistores en serie. Puesto que, como dijimos en la sección 3.8.1, los transistores PMOS tienen más o menos la mitad de la fuerza de dirección de los transistores NMOS, el tamaño efectivo de los tres transistores PMOS en serie debe hacerse aproximadamente del doble del de un transistor NMOS. Por tanto,

$$L_p = L_n \times 3 \times 2 = 6L_n$$

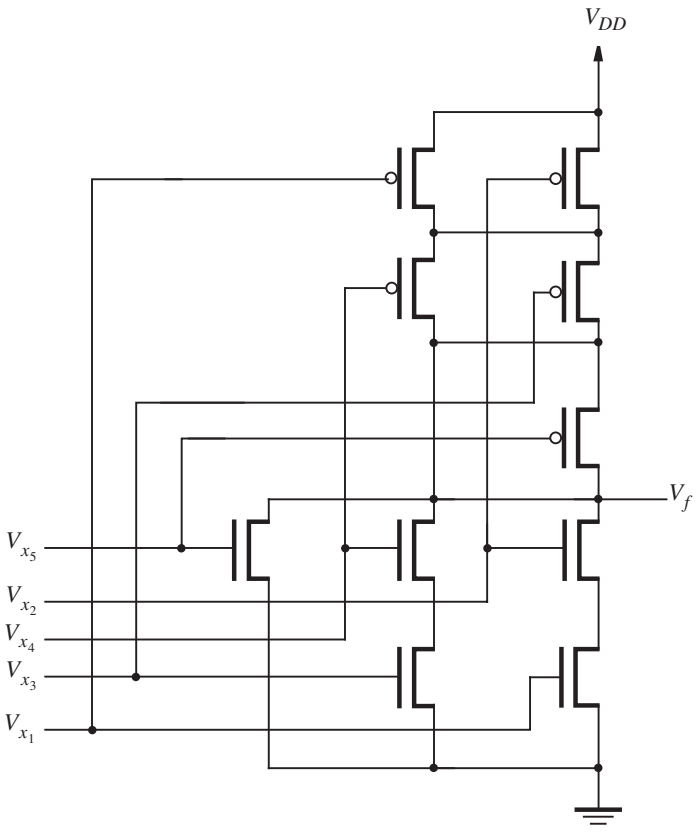


Figura 3.71 Circuito para el ejemplo 3.9.

Ejemplo 3.11

Problema: En la sección 3.8.5 dijimos que el tiempo necesario para cargar un capacitor está dado por

$$t_p = \frac{C\Delta V}{I}$$

Derive esta expresión.

Solución: Como afirmamos en la sección 3.8.5, el voltaje a través de un capacitor no puede cambiar instantáneamente. En la figura 3.50*a*, conforme V_f se carga desde 0 voltios hasta V_{DD}, el voltaje cambia de acuerdo con la ecuación

$$V_f = \frac{1}{C}\int_0^{\infty} i(t)\,dt$$

En esta expresión, la variable independiente t es tiempo, e $i(t)$ representa el flujo de corriente instantáneo a través del capacitor en el tiempo t. Al diferenciar ambos miembros de esta expresión

respecto al tiempo y reordenar términos se obtiene

$$i(t) = C\frac{dV_f}{dt}$$

Para el caso donde I es constante se tiene

$$\frac{I}{C} = \frac{\Delta V}{\Delta t}$$

Por tanto,

$$\Delta t = t_p = \frac{C\Delta V}{I}$$

Ejemplo 3.12 **Problema:** En la explicación de la figura 3.50*a*, en la sección 3.8.6, dijimos que un capacitor, C, cargado al voltaje $V_f = V_{DD}$, almacena una cantidad de energía igual a $CV_{DD}^2/2$. Derive esta expresión.

Solución: Como demostramos en el ejemplo 3.11, el flujo de corriente a través de un capacitor que se carga, C, está relacionado con la tasa de cambio del voltaje a través del capacitor, de acuerdo con

$$i(t) = C\frac{dV_f}{dt}$$

La potencia instantánea disipada en el capacitor es

$$P = i(t) \times V_f$$

Ya que la energía se define como la potencia utilizada durante un periodo de tiempo es posible calcular la energía, E_C, almacenada en el capacitor conforme V_f cambia de 0 a V_{DD}, al integrar la potencia instantánea sobre el tiempo del modo siguiente

$$E_C = \int_0^\infty i(t)V_f\,dt$$

Al sustituir la expresión anterior para $i(t)$ se obtiene

$$\begin{aligned} E_C &= \int_0^\infty C\frac{dV_f}{dt}V_f\,dt \\ &= C\int_0^{V_{DD}} V_f\,dV_f \\ &= \frac{1}{2}CV_{DD}^2 \end{aligned}$$

Ejemplo 3.13 **Problema:** En la tecnología original NMOS, el dispositivo de subida era un MOSFET de n canales. Pero la mayor parte de los circuitos integrados fabricados hoy día usa tecnología CMOS. Por tanto, es conveniente implementar el resistor de subida con un transistor PMOS, como se

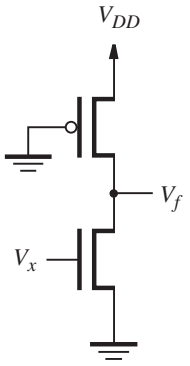


Figura 3.72 Inversor seudoNMOS.

muestra en la figura 3.72. Tal circuito recibe el nombre de circuito *seudoNMOS*. El dispositivo de subida se llama *transistor PMOS "débil"* porque tiene una razón W/L pequeña.

Cuando $V_x = V_{DD}$, V_f tiene un valor bajo. El transistor NMOS opera en la región de tríodo, mientras que el PMOS limita el flujo de corriente porque funciona en la región de saturación. La corriente a través de los transistores NMOS y PMOS debe ser igual y está dada por las ecuaciones 3.1 y 3.2. Demuestre que el voltaje de salida baja, $V_f = V_{OL}$, está dado por

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

donde k_p y k_n, denominados *factores de ganancia*, dependen del tamaño de los transistores PMOS y NMOS, respectivamente. Dichos factores se definen como $k_p = k'_p W_p/L_p$ y $k_n = k'_n W_n/L_n$.

Solución: Por simplicidad se supondrá que las magnitudes de los voltajes umbral para ambos transistores son iguales, de modo que

$$V_T = V_{TN} = -V_{TP}$$

El transistor PMOS opera en la región de saturación, por lo que la corriente que fluye a través de él está dada por

$$\begin{aligned} I_D &= \frac{1}{2}k'_p\frac{W_p}{L_p}(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(-V_{DD} - V_{TP})^2 \\ &= \frac{1}{2}k_p(V_{DD} - V_T)^2 \end{aligned}$$

De manera similar, el transistor NMOS opera en la región tríodo, y su flujo de corriente se define por

$$I_D = k_n' \frac{W_n}{L_n}\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right]$$
$$= k_n\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right]$$
$$= k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right]$$

Puesto que sólo existe una trayectoria para que la corriente fluya es posible igualar las corrientes que pasan por los transistores NMOS y PMOS, y resolver para el voltaje V_f.

$$k_p(V_{DD} - V_T)^2 = 2k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right]$$
$$k_p(V_{DD} - V_T)^2 - 2k_n(V_{DD} - V_T)V_f + k_nV_f^2 = 0$$

Esta ecuación cuadrática puede resolverse mediante la fórmula estándar, con los parámetros

$$a = k_n,\ b = -2k_n(V_{DD} - V_T),\ c = k_p(V_{DD} - V_T)^2$$

lo que produce

$$V_f = \frac{-b}{2a} \pm \sqrt{\frac{b^2}{4a^2} - \frac{c}{a}}$$
$$= (V_{DD} - V_T) \pm \sqrt{(V_{DD} - V_T)^2 - \frac{k_p}{k_n}(V_{DD} - V_T)^2}$$
$$= (V_{DD} - V_T)\left[1 \pm \sqrt{1 - \frac{k_p}{k_n}}\right]$$

Sólo una de estas dos soluciones es válida, porque se comenzó con la suposición de que el transistor NMOS está en la región de tríodo mientras que el PMOS se halla en la región de saturación. Por tanto

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

Ejemplo 3.14 **Problema:** Para el circuito de la figura 3.72, suponga los valores $k_n' = 60\ \mu\text{A/V}^2$, $k_p' = 0.4\,k_n'$, $W_n/L_n = 2.0\ \mu\text{m}/0.5\ \mu\text{m}$, $W_p/L_p = 0.5\ \mu\text{m}/0.5\ \mu\text{m}$, $V_{DD} = 5$ V y $V_T = 1$ V. Cuando $V_x = V_{DD}$, calcule lo siguiente:

a) La corriente estática, $I_{estát}$.

b) La resistencia de encendido del transistor NMOS.

c) V_{OL}.

d) La potencia estática disipada en el inversor.

e) La resistencia de encendido del transistor PMOS.

f) Suponga que el inversor se usa para dirigir una carga capacitiva de 70 fF. Con la ecuación 3.4, calcule los retrasos de propagación de bajo a alto y de alto a bajo.

Solución: *a*) El transistor PMOS está saturado; por tanto

$$\begin{aligned} I_{estát} &= \frac{1}{2}k_p'\frac{W_p}{L_p}(V_{DD} - V_T)^2 \\ &= 12\ \frac{\mu\text{A}}{\text{V}^2} \times 1 \times (5\ \text{V} - 1\ \text{V})^2 = 192\ \mu\text{A} \end{aligned}$$

b) Al usar la ecuación 3.3,

$$\begin{aligned} R_{DS} &= 1/\left[k_n'\frac{W_n}{L_n}(V_{GS} - V_T)\right] \\ &= 1/\left[0.060\ \frac{\text{mA}}{\text{V}^2} \times 4 \times (5\ \text{V} - 1\ \text{V})\right] = 1.04\ \text{k}\Omega \end{aligned}$$

c) Si se emplea la expresión derivada en el ejemplo 3.13 se obtiene

$$k_p = k_p'\frac{W_p}{L_p} = 24\ \frac{\mu\text{A}}{\text{V}^2}$$

$$k_n = k_n'\frac{W_n}{L_n} = 240\ \frac{\mu\text{A}}{\text{V}^2}$$

$$\begin{aligned} V_{OL} = V_f &= (5\ \text{V} - 1\ \text{V})\left[1 - \sqrt{1 - \frac{24}{240}}\right] \\ &= 0.21\ \text{V} \end{aligned}$$

d)

$$\begin{aligned} P_D &= I_{estát} \times V_{DD} \\ &= 192\ \mu\text{A} \times 5\ \text{V} = 960\ \mu\text{W} \approx 1\ \text{mW} \end{aligned}$$

e)

$$\begin{aligned} R_{SDP} &= V_{SD}/I_{SD} \\ &= (V_{DD} - V_f)/I_{estát} \\ &= (5\ \text{V} - 0.21\ \text{V})/0.192\ \text{mA} = 24.9\ \text{k}\Omega \end{aligned}$$

f) El retraso de propagación de bajo a alto es

$$\begin{aligned} t_{pLH} &= \frac{1.7C}{k_p'\frac{W_p}{L_p}V_{DD}} \\ &= \frac{1.7 \times 70\ \text{fF}}{24\ \frac{\mu\text{A}}{\text{V}^2} \times 1 \times 5\ \text{V}} = 0.99\ \text{ns} \end{aligned}$$

El retraso de propagación de alto a bajo es

$$t_{pHL} = \frac{1.7C}{k_n' \frac{W_n}{L_n} V_{DD}}$$
$$= \frac{1.7 \times 70 \text{ fF}}{60 \frac{\mu\text{A}}{\text{V}^2} \times 4 \times 5 \text{ V}} = 0.1 \text{ ns}$$

Ejemplo 3.15 **Problema:** En la figura 3.69 mostramos una solución al problema de la disipación de potencia estática cuando se usan transistores de paso NMOS. Suponga que de este circuito se elimina el transistor de subida PMOS. Suponga los parámetros $k_n' = 60\ \mu\text{A/V}^2$, $k_p' = 0.5 \times k_n'$, $W_n/L_n = 2.0\ \mu\text{m}/0.5\ \mu\text{m}$, $W_p/L_p = 4.0\ \mu\text{m}/0.5\ \mu\text{m}$, $V_{DD} = 5$ V y $V_T = 1$ V. Para $V_B = 3.5$ V, calcule lo siguiente:

a) La corriente estática $I_{estát}$.

b) El voltaje V_f en la salida del inversor.

c) La disipación de potencia estática en el inversor.

d) Si un chip contiene 250 000 inversores usados de esta forma, encuentre la disipación total de potencia estática.

Solución: *a*) Si suponemos que el transistor PMOS opera en la región de saturación, entonces el flujo de corriente por el inversor se define por

$$I_{estát} = \frac{1}{2} k_p' \frac{W_p}{L_p} (V_{GS} - V_{Tp})^2$$
$$= 120 \frac{\mu\text{A}}{\text{V}^2} \times ((3.5 \text{ V} - 5 \text{ V}) + 1 \text{ V})^2 = 30\ \mu\text{A}$$

b) Puesto que la corriente estática $I_{estát}$ que fluye a través del transistor PMOS también fluye por el transistor NMOS, si suponemos que el transistor NMOS opera en la región de tríodo, se tiene que

$$I_{estát} = k_n' \frac{W_n}{L_n} \left[(V_{GS} - V_{Tn}) V_{DS} - \frac{1}{2} V_{DS}^2 \right]$$
$$30\ \mu\text{A} = 240 \frac{\mu\text{A}}{\text{V}^2} \times \left[2.5 \text{ V} \times V_f - \frac{1}{2} V_f^2 \right]$$
$$1 = 20 V_f - 4 V_f^2$$

La resolución de esta ecuación cuadrática produce $V_f = 0.05$ V. Nótese que el voltaje de salida V_f satisface la suposición de que el transistor PMOS opera en la región de saturación, mientras que el NMOS lo hace en la región de tríodo.

c) La potencia estática disipada en el inversor es

$$P_S = I_{estát} \times V_{DD} = 30\ \mu\text{A} \times 5 \text{ V} = 150\ \mu\text{W}$$

d) La potencia estática disipada por 250 000 inversores es

$$250\,000 \times P_S = 37.5 \text{ W}$$

PROBLEMAS

Al final del libro se presentan las respuestas a los problemas marcados con asterisco.

3.1 Considere el circuito mostrado en la figura P3.1.
a) Elabore la tabla de verdad para la función *f*.
b) Si cada compuerta del circuito se implementa como una compuerta CMOS, ¿cuántos transistores se necesitan?

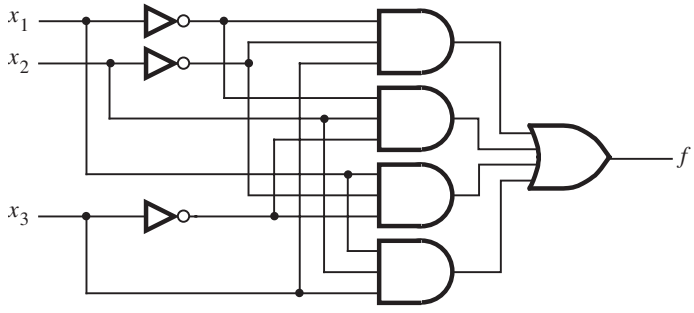


Figura P3.1 Circuito CMOS en suma de productos.

3.2 *a*) Demuestre que el circuito de la figura P3.2 equivale funcionalmente al de la figura P3.1.
b) ¿Cuántos transistores se necesitan para construir este circuito CMOS?

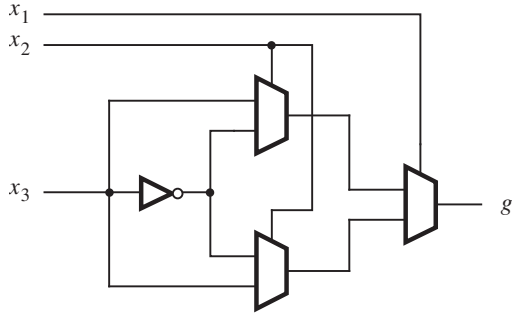


Figura P3.2 Circuito CMOS construido con multiplexores.

3.3 *a*) Demuestre que el circuito de la figura P3.3 equivale funcionalmente al de la figura P3.2.
b) ¿Cuántos transistores se necesitan para construir este circuito CMOS si cada compuerta XOR se implementa usando el circuito de la figura 3.61*d*?

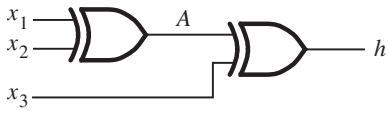


Figura P3.3 Circuito para el problema 3.3.

***3.4** En la sección 3.8.8 dijimos que es posible construir una compuerta AND CMOS de seis entradas con dos compuertas AND de tres entradas y una compuerta AND de dos. Este enfoque requiere 22 transistores. Muestre cómo usar sólo compuertas NAND y NOR CMOS para construir la compuerta AND de seis entradas y calcule el número necesario de transistores. (Sugerencia: aplique el teorema de DeMorgan.)

3.5 Repita el problema 3.4 para una compuerta OR CMOS de ocho entradas.

3.6 *a*) Elabore la tabla de verdad para el circuito CMOS de la figura P3.4.
b) Derive una expresión canónica en suma de productos para la tabla de verdad del inciso *a*). ¿Cuántos transistores se precisan para construir un circuito que represente la forma canónica si sólo se usan compuertas AND, OR y NOT?

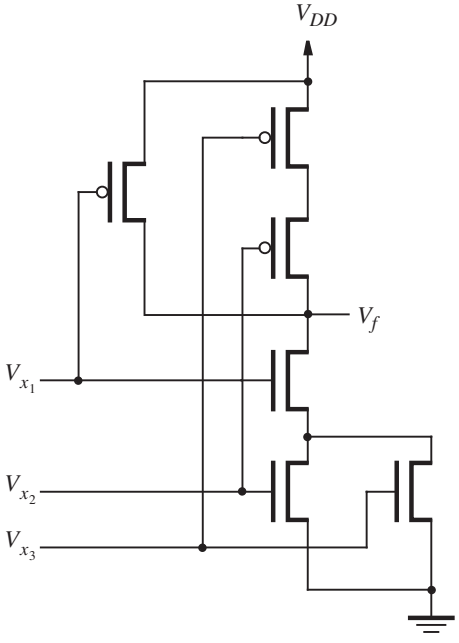


Figura P3.4 Circuito CMOS de tres entradas.

3.7 *a*) Elabore la tabla de verdad para el circuito CMOS de la figura P3.5.
b) Derive la expresión más simple de suma de productos para la tabla de verdad del inciso *a*). ¿Cuántos transistores se necesitan para construir el circuito de suma de productos usando compuertas AND, OR y NOT CMOS?

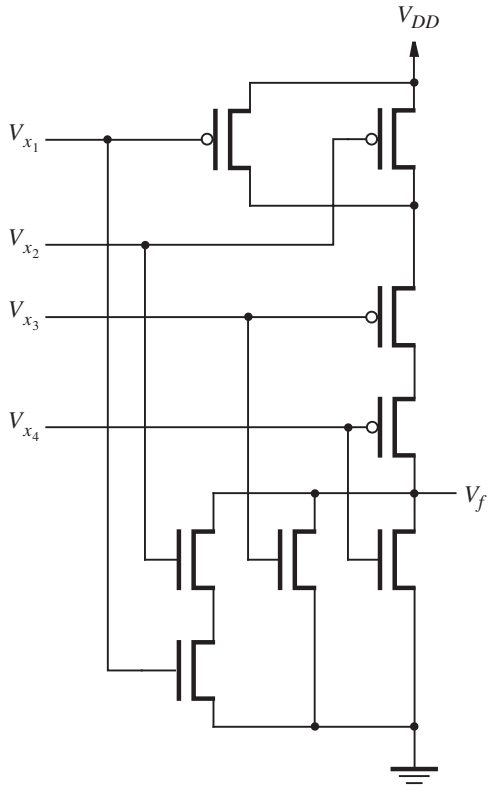


Figura P3.5 Circuito CMOS de cuatro entradas.

***3.8** En la figura P3.6 se muestra la mitad de un circuito CMOS. Derive la otra mitad que contenga los transistores PMOS.

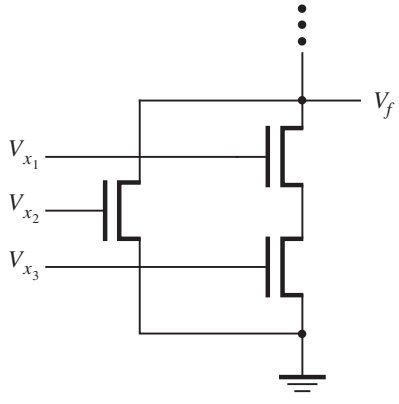


Figura P3.6 PDN en un circuito CMOS.

3.9 En la figura P3.7 se muestra la mitad de un circuito CMOS. Derive la otra mitad que contenga los transistores NMOS.

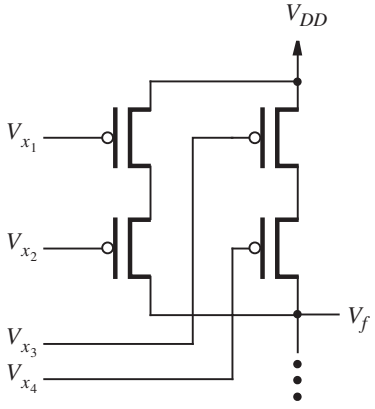


Figura P3.7 PDN en un circuito CMOS.

3.10 Derive una compuerta compleja CMOS para la función lógica $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 10)$.

3.11 Derive una compuerta compleja CMOS para la función lógica $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 6, 8, 9, 10, 12, 14)$.

***3.12** Derive una compuerta compleja CMOS para la función lógica $f = xy + xz$. Use el menor número de transistores posible. (Sugerencia: Considere $\overline{f}$).

3.13 Derive una compuerta compleja CMOS para la función lógica $f = xy + xz + yz$. Use el menor número de transistores posible. (Sugerencia: Considere $\overline{f}$).

***3.14** Para un transistor NMOS, suponga que $k_n' = 20\ \mu A/V^2$, $W/L = 2.5\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V y $V_T = 1$ V. Calcule

a) I_D cuando $V_{DS} = 5$ V

b) I_D cuando $V_{DS} = 0.2$ V

3.15 Para un transistor PMOS, suponga que $k_p' = 10\ \mu A/V^2$, $W/L = 2.5\ \mu m/0.5\ \mu m$, $V_{GS} = -5$ V y $V_T = -1$ V. Calcule

a) I_D cuando $V_{DS} = -5$ V

b) I_D cuando $V_{DS} = -0.2$ V

3.16 Para un transistor NMOS, suponga que $k_n' = 20\ \mu A/V^2$, $W/L = 5.0\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V y $V_T = 1$ V. Calcule R_{DS} para pequeños V_{DS}.

***3.17** Para un transistor NMOS, suponga que $k_n' = 40\ \mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = 3.3$ V y $V_T = 0.66$ V. Calcule R_{DS} para pequeños V_{DS}.

3.18 Para un transistor PMOS, suponga que $k'_p = 10\ \mu A/V^2$, $W/L = 5.0\ \mu m/0.5\ \mu m$, $V_{GS} = -5$ V y $V_T = -1$ V. Para $V_{DS} = -4.8$ V, calcule R_{DS}.

3.19 Para un transistor PMOS, suponga que $k'_p = 16\ \mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = -3.3$ V y $V_T = -0.66$ V. Para $V_{DS} = -3.2$ V, calcule R_{DS}.

3.20 En el ejemplo 3.13 mostramos cómo calcular niveles de voltaje en un inversor seudoNMOS. En la figura P3.8 se describe un inversor seudoPMOS. En esta tecnología se usa un transistor NMOS débil para implementar un resistor de bajada.

Cuando $V_x = 0$, V_f tiene un valor alto. El transistor PMOS opera en la región de tríodo, mientras que el NMOS limita el flujo de corriente porque opera en la de saturación. La corriente que pasa por los transistores PMOS y NMOS ha de ser la misma y está dada por las ecuaciones 3.1 y 3.2. Encuentre una expresión para el voltaje de salida alto, $V_f = V_{OH}$, en términos de V_{DD}, V_T, k_p y k_n, donde k_p y k_n son factores de ganancia como se definió en el ejemplo 3.13.

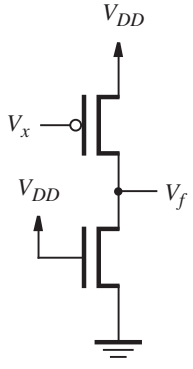


Figura P3.8 Inversor seudoNMOS.

3.21 Para el circuito de la figura P3.8, suponga los valores $k'_n = 60\ \mu A/V^2$, $k'_p = 0.4\,k'_n$, $W_n/L_n = 0.5\ \mu m/0.5\ \mu m$, $W_p/L_p = 4.0\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V y $V_T = 1$ V. Cuando $V_x = 0$, calcule lo siguiente:

a) La corriente estática, $I_{estát}$

b) La resistencia de encendido del transistor PMOS

c) V_{OH}

d) La potencia estática disipada en el inversor

e) La resistencia de encendido del transistor NMOS

f) Suponga que el inversor se usa para dirigir una carga capacitiva de 70 fF. Con la ecuación 3.4, calcule los retrasos de propagación de bajo a alto y de alto a bajo.

3.22 Repita el problema 3.21 y suponga que el tamaño del transistor NMOS cambia a $W_n/L_n = 4.0\ \mu$m/0.5 μm.

3.23 En el ejemplo 3.13 (véase la figura 3.72) se muestra que en la tecnología seudoNMOS el dispositivo de subida se implementa con un transistor PMOS. Repita este problema para una compuerta NAND construida con tecnología seudoNMOS. Asuma que ambos transistores NMOS en la compuerta tienen los mismos parámetros, como se especifica en el ejemplo 3.14.

3.24 Repita el problema 3.23 para una compuerta NOR seudoNMOS.

***3.25** *a*) Para $Vm = 4$ V, $V_{OH} = 4.5$, $V_{IL} = 1$V, $V_{OL} = 0.3$ V y $V_{DD} = 5$ V, calcule los márgenes de ruido NM_H y NM_L.
b) Considere una compuerta NAND de ocho entradas construida con tecnología NMOS. Si la caída de voltaje a través de cada transistor es 0.1 V, ¿cuál es V_{OL}? ¿Cuál es el correspondiente NM_L si se usan los otros parámetros del inciso *a*)?

3.26 En condiciones de estado estacionario, para una compuerta NAND CMOS de *n* entradas, ¿cuáles son los niveles de voltaje de V_{OL} y V_{OH}? Explique.

3.27 Para un inversor CMOS, suponga que la capacitancia de carga es $C = 150$ fF y $V_{DD} = 5$ V. El inversor realiza ciclos a través de los niveles de voltaje alto y bajo a una tasa promedio de $f = 75$ MHz.

a) Calcule la potencia dinámica disipada en el inversor.

b) Para un chip que contiene el equivalente de 250 000 inversores, calcule la potencia dinámica total disipada si 20% de las compuertas cambia valores en cualquier tiempo dado.

***3.28** Repita el problema 3.27 para $C = 120$ fF, $V_{DD} = 3.3$ V y $f = 125$ MHz.

3.29 En un inversor CMOS, suponga que $k'_n = 20\ \mu$A/V^2, $k'_p = 0.4 \times k'_n$, $W_n/L_n = 5.0\ \mu$m/0.5 μm, $W_p/L_p = 5.0\ \mu$m/0.5 μm y $V_{DD} = 5$ V. El inversor dirige una capacitancia de carga de 150 fF.
a) Encuentre el retraso de propagación de alto a bajo.
b) Encuentre el retraso de baja propagación de bajo a alto.
c) ¿Cuáles deben ser las dimensiones del transistor PMOS de tal modo que los retrasos de propagación de bajo a alto y de alto a bajo sean iguales? Ignore el efecto del tamaño del transistor PMOS en la capacitancia de carga del inversor.

3.30 Repita el problema 3.29 para los parámetros $k'_n = 40\ \mu$A/V^2, $k'_p = 0.4 \times k'_n$, $W_n/L_n = W_p/L_p = 3.5\ \mu$m/0.35 μm y $V_{DD} = 3.3$ V.

3.31 En un inversor CMOS, suponga que $W_n/L_n = 2$ y $W_p/L_p = 4$. Para una compuerta NAND CMOS, calcule las razones requeridas W/L de los transistores NMOS y PMOS tales que la corriente disponible en la compuerta para dirigir la salida tanto a bajo como a alto sea igual a la del inversor.

***3.32** Repita el problema 3.31 para una compuerta NOR CMOS.

3.33 Repita el problema 3.31 para la compuerta compleja CMOS de la figura 3.16. Debe elegir el tamaño de los transistores de tal modo que, en el peor de los casos, la corriente disponible sea al menos tan grande como en el inversor.

3.34 Repita el problema 3.31 para la compuerta compleja CMOS de la figura 3.17.

3.35 En la figura 3.69 mostramos una solución al problema de disipación de potencia estática cuando se usan transistores de paso NMOS. Suponga que el transistor de subida PMOS se quita de este circuito. Asuma los parámetros $k_n' = 60\,\mu\text{A/V}^2$, $k_p' = 0.4 \times k_n'$, $W_n/L_n = 1.0\,\mu\text{m}/0.25\,\mu\text{m}$, $W_p/L_p = 2.0\,\mu\text{m}/0.25\,\mu\text{m}$, $V_{DD} = 2.5$ V y $V_T = 0.6$ V. Para $V_B = 1.6$ V, calcule lo siguiente:
a) La corriente estática, $I_{estát}$
b) El voltaje, V_f, en la salida del inversor
c) La disipación de potencia estática en el inversor
d) Si un chip contiene 500 000 inversores utilizados de esta forma, encuentre la disipación total de potencia estática.

3.36 Con el estilo de dibujo de la figura 3.66, trace una ilustración de un PLA programado para implementar $f_1(x_1, x_2, x_3) = \sum m(1, 2, 4, 7)$. El PLA debe tener las entradas $x_1, \ldots, x_3$; los términos producto $P_1, \ldots, P_4$; y las salidas f_1 y f_2.

3.37 Con el estilo de dibujo de la figura 3.66, trace una ilustración de un PLA programado para implementar $f_1(x_1, x_2, x_3) = \sum m(0, 3, 5, 6)$. El PLA debe tener las entradas $x_1, \ldots, x_3$; los términos producto $P_1, \ldots, P_4$; y las salidas f_1 y f_2.

3.38 Muestre cómo se puede realizar la función f_1 del problema 3.36 en un PLA del tipo mostrado en la figura 3.65. Dibuje una ilustración de tal PLA programado para implementar f_1. El PLA debe tener las entradas $x_1, \ldots, x_3$; los términos suma $S_1, \ldots, S_4$; y las salidas f_1 y f_2.

3.39 Muestre cómo puede realizarse la función f_1 del problema 3.37 en un PLA del tipo mostrado en la figura 3.65. Dibuje una ilustración de tal PLA programado para implementar f_1. El PLA debe tener las entradas $x_1, \ldots, x_3$; los términos suma $S_1, \ldots, S_4$; y las salidas f_1 y f_2.

3.40 Repita el problema 3.38 con el estilo de dibujo PLA mostrado en la figura 3.63.

3.41 Repita el problema 3.39 con el estilo de dibujo PLA mostrado en la figura 3.63.

3.42 Suponga que f_1 se implementa como se describió en el problema 3.36 y enumere todas las otras posibles funciones lógicas que pueden realizarse con la salida f_2 en el PLA.

3.43 Suponga que f_1 se implementa como se describió en el problema 3.37 y enumere todas las otras posibles funciones lógicas que pueden realizarse con la salida f_2 en el PLA.

3.44 Considere la función $f(x_1, x_2, x_3) = x_1\overline{x}_2 + x_1x_3 + x_2\overline{x}_3$. Indique un circuito que use cinco tablas de consulta (LUT) de dos entradas para implementar esta expresión. Como se muestra en la figura 3.39, elabore la tabla de verdad implementada en cada LUT. No necesita mostrar los cables del FPGA.

***3.45** Considere la función $f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$. Indique cómo se puede realizar con dos LUT de dos entradas. Como se muestra en la figura 3.39, proporcione la tabla de verdad implementada en cada LUT. No necesita mostrar los cables del FPGA.

3.46 Suponga la función $f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$, entonces una implementación directa en un FPGA con LUT de tres entradas requiere cuatro LUT. Muestre cómo puede hacerse usando solamente tres LUT de tres entradas. Etiquete la salida de cada LUT con una expresión que represente la función lógica que implementa.

3.47 Para *f* en el problema 3.46, muestre un circuito de LUT de dos entradas que realice la función. Debe usar exactamente siete LUT de dos entradas. Etiquete la salida de cada LUT con una expresión que represente la función lógica que implementa.

3.48 En la figura 3.39 se muestra un FPGA programado para implementar una función. En la figura se observa un pin usado para la función *f* y varios pines que no se utilizan. Sin cambiar la programación de algún interruptor que esté *encendido* en el FPGA de la figura, enumere otras 10 funciones lógicas, además de *f*, que puedan implementarse en los pines no usados.

3.49 Suponga que un arreglo de compuertas contiene el tipo de celdas lógicas descrito en la figura P3.9. Las entradas $in_1, \ldots, in_7$ pueden conectarse a 1, a 0 o a cualquier señal lógica.

a) Muestre cómo puede usarse la celda lógica para realizar $f = x_1x_2 + x_3$.

b) Muestre cómo puede usarse la celda lógica para realizar $f = x_1x_3 + x_2x_3$.

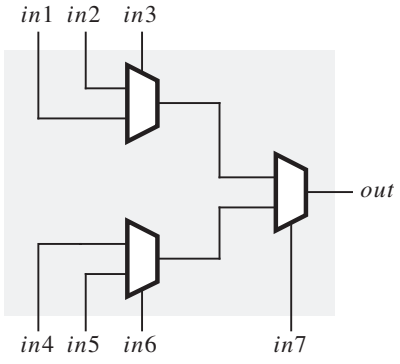


Figura P3.9 Celda lógica de arreglo de compuertas.

3.50 Suponga que existe un arreglo de compuertas en el que la celda lógica empleada es una compuerta NAND de tres entradas. Las entradas a cada compuerta NAND pueden conectarse a 1, a 0 o a cualquier señal lógica. Muestre cómo pueden realizarse las funciones lógicas siguientes en el arreglo de compuertas. (Sugerencia: Aplique el teorema de DeMorgan.)

a) $f = x_1x_2 + x_3$

b) $f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1$

3.51 Escriba código de VHDL para representar la función

$$f = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_4 + \overline{x}_1x_2x_3 + x_1x_2x_3$$

a) Use sus herramientas CAD para implementar *f* en algún tipo de chip, digamos un CPLD. Demuestre la expresión lógica generada para *f* por las herramientas. Use simulación de tiempo para determinar el tiempo necesario para un cambio en las entradas x_1, x_2 o x_3 para que se propaguen a la salida *f*.

b) Repita el inciso *a*) con un chip diferente, digamos un FPGA, para la implementación del circuito.

3.52 Repita el problema 3.51 para la función

$$f = (x_1 + x_2 + \overline{x}_4) \cdot (\overline{x}_2 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$$

3.53 Repita el problema 3.51 para la función

$$f(x_1, \ldots, x_7) = x_1 x_3 \overline{x}_6 + x_1 x_4 x_5 \overline{x}_6 + x_2 x_3 x_7 + x_2 x_4 x_5 x_7$$

3.54 ¿Qué compuerta lógica realiza el circuito de la figura P3.10? ¿Tiene este circuito inconvenientes mayores? ¿Cuáles?

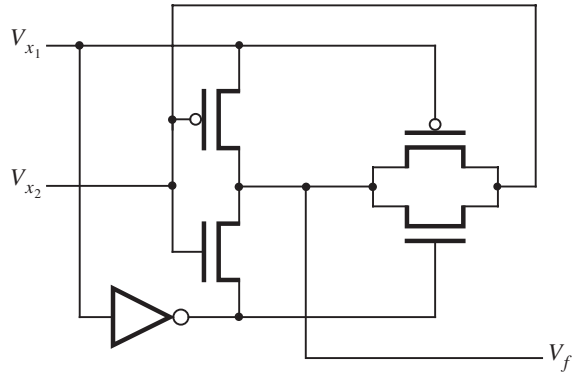


Figura P3.10 Circuito para el problema 3.54.

***3.55** ¿Qué compuerta lógica realiza el circuito de la figura P3.11? ¿Tiene este circuito inconvenientes mayores? ¿Cuáles?

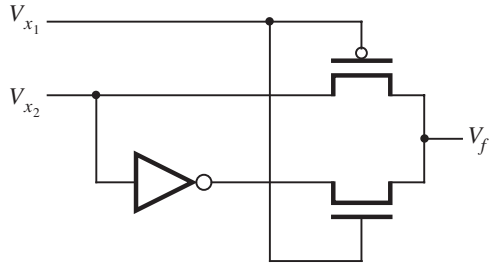


Figura P3.11 Circuito para el problema 3.55.

BIBLIOGRAFÍA

1. A. S. Sedra y K. C. Smith, *Microelectronic Circuits*, 5a. ed. (Oxford University Press: Nueva York, 2003).
2. J. M. Rabaey, *Digital Intergrated Circuits* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. Texas Instruments, *Logic Products Selection Guide and Databook CD-ROM*, 1997.
4. National Semiconductor, *VHC/VHCT Advanced CMOS Logic Databook*, 1993.
5. Motorola, *CMOS Logic Databook*, 1996.
6. Toshiba America Electronic Components, *TC74VHC/VHCT Series CMOS Logic Databook*, 1994.
7. Integrated Devices Technology, *High Performance Logic Databook*, 1994.
8. J. F. Wakerly, *Digital Design Principles and Practices*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
9. M. M. Mano, *Digital Design* 3a. ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
10. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
11. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
12. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

capítulo

4

Implementación optimizada de funciones lógicas

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- Síntesis de las funciones lógicas
- Análisis de los circuitos lógicos
- Técnicas para derivar implementaciones de costo mínimo de las funciones lógicas
- Representación gráfica de funciones lógicas mediante mapas de Karnaugh
- Representación cúbica de funciones lógicas
- El uso de las herramientas CAD y de VHDL para implementar funciones lógicas

En el capítulo 2 mostramos que la manipulación algebraica permite hallar la implementación de menor costo de las funciones lógicas. El propósito de ese capítulo era introducir los conceptos básicos en el proceso de síntesis. Probablemente el lector esté convencido de que es fácil derivar una realización directa de una función lógica en una forma canónica, pero no es del todo obvio cómo elegir y aplicar los teoremas y propiedades de la sección 2.5 a fin de hallar un circuito de costo mínimo. De hecho, la manipulación algebraica es más bien tediosa y muy impráctica para funciones de muchas variables.

Si se usan herramientas CAD para diseñar circuitos lógicos, la tarea de minimizar el costo de implementación no recae en el diseñador; las herramientas realizan automáticamente la optimación necesaria. Aun así, es esencial saber algo acerca de este proceso. La mayor parte de las herramientas CAD tiene muchas características y opciones que quedan bajo el control del usuario. Para saber cuándo y cómo aplicarlas, éste debe conocer qué hacen las herramientas.

En este capítulo expondremos algunas de las técnicas de optimación incluidas en las herramientas CAD y mostraremos cómo automatizarlas. En primer lugar explicaremos un enfoque gráfico, conocido como *mapa de Karnaugh*, que ofrece una forma elegante de derivar manualmente implementaciones de costo mínimo de funciones lógicas simples. Aunque no está disponible para implementarlo en las herramientas CAD, sí ilustra varios conceptos clave. Mostraremos cómo diseñar circuitos de dos y varios niveles. Luego describiremos una representación cúbica de las funciones lógicas, que está disponible para emplearla en las herramientas CAD. También proseguiremos con nuestra explicación del lenguaje VHDL.

4.1 MAPA DE KARNAUGH

En la sección 2.6 vimos que la clave para hallar una expresión de costo mínimo para una función lógica consiste en reducir el número de términos producto (o suma) necesarios en la expresión. Ello se logra mediante la aplicación de la propiedad combinatoria 14*a* (o 14*b*) cuan juiciosamente sea posible. El enfoque de mapa de Karnaugh brinda una forma sistemática de realizar esta optimación. Para entender cómo funciona será útil revisar el enfoque algebraico del capítulo 2. Considérese la función f de la figura 4.1. La expresión canónica en suma de productos para f consta de los mintérminos m_0, m_2, m_4, m_5 y m_6, de modo que

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

La propiedad combinatoria 14*a* permite sustituir dos mintérminos que difieren sólo en el valor de una variable con un solo término producto que no incluye esa variable. Por ejemplo, tanto m_0 como m_2 incluyen $\overline{x}_1$ y $\overline{x}_3$, pero difieren en el valor de x_2 porque m_0 incluye $\overline{x}_2$ mientras que m_2 incluye x_2. Por ende

$$\begin{aligned}\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 &= \overline{x}_1(\overline{x}_2 + x_2)\overline{x}_3\\ &= \overline{x}_1 \cdot 1 \cdot \overline{x}_3\\ &= \overline{x}_1\overline{x}_3\end{aligned}$$

Número de fila	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figura 4.1 La función $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

Por tanto, m_0 y m_2 pueden sustituirse por el término producto $\overline{x}_1\overline{x}_3$. De manera similar, m_4 y m_6 difieren sólo en el valor de x_2 y pueden combinarse usando

$$\begin{aligned} x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 &= x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= x_1 \cdot 1 \cdot \overline{x}_3 \\ &= x_1\overline{x}_3 \end{aligned}$$

Ahora los dos términos recién generados, $\overline{x}_1\overline{x}_3$ y $x_1\overline{x}_3$ pueden combinarse todavía más como

$$\begin{aligned} \overline{x}_1\overline{x}_3 + x_1\overline{x}_3 &= (\overline{x}_1 + x_1)\overline{x}_3 \\ &= 1 \cdot \overline{x}_3 \\ &= \overline{x}_3 \end{aligned}$$

Estos pasos de optimación indican que podemos sustituir los cuatro mintérminos m_0, m_2, m_4 y m_6 con el término producto $\overline{x}_3$. En otras palabras, los mintérminos m_0, m_2, m_4 y m_6 están todos *incluidos* en el término $\overline{x}_3$. El mintérmino restante en f es m_5. Puede combinarse con m_4, lo que produce

$$x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 = x_1\overline{x}_2$$

Recuérdese que el teorema 7*b* de la sección 2.5 postula que

$$m_4 = m_4 + m_4$$

lo que significa que es posible usar el mintérmino m_4 dos veces: para combinarse con los mintérminos m_0, m_2 y m_6 para producir el término $\overline{x}_3$ como se explicó anteriormente, y también para combinarse con m_5 para producir el término $x_1\overline{x}_2$.

Ahora hemos dado cuenta de todos los mintérminos en f; por tanto, las cinco combinaciones de entrada para las que $f = 1$ están cubiertas por la expresión de costo mínimo

$$f = \overline{x}_3 + x_1\overline{x}_2$$

La expresión tiene el término producto $\overline{x}_3$ porque $f = 1$ cuando $x_3 = 0$ independientemente de los valores de x_1 y x_2. Los cuatro mintérminos m_0, m_2, m_4 y m_6 representan todos los mintérminos posibles para los que $x_3 = 0$; incluyen las cuatro combinaciones, 00, 01, 10 y 11, de las variables x_1 y x_2. Por ende, si $x_3 = 0$, entonces está garantizado que $f = 1$. Esto puede no ser fácil de ver directamente a partir de la tabla de verdad de la figura 4.1, pero es obvio si escribimos agrupadas las combinaciones correspondientes:

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

De forma similar, si se observan m_4 y m_5 como un grupo de dos

	x_1	x_2	x_3
m_4	1	0	0
m_5	1	1	1

es claro que cuando $x_1 = 1$ y $x_2 = 0$, entonces $f = 1$ independientemente de los valores de x_3.

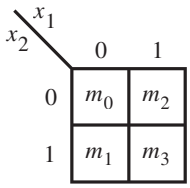
La explicación anterior sugiere que sería ventajoso elaborar un método que permita el descubrimiento rápido de grupos de mintérminos para los que $f = 1$ y que puedan combinarse en términos individuales. El mapa de Karnaugh es un vehículo útil para tal propósito.

El *mapa de Karnaugh* [1] es una alternativa a la forma de tabla de verdad para representar una función. Consta de *celdas* que corresponden a las filas de la tabla de verdad. Considérese el ejemplo de dos variables de la figura 4.2. En el inciso *a*) se muestra la forma en tabla de verdad, donde cada una de las cuatro filas se identifica mediante un mintérmino. En el inciso *b*) se observa el mapa de Karnaugh, que tiene cuatro celdas. Las columnas están etiquetadas con el valor de x_1, y las filas con el de x_2. Esta forma de etiquetar conduce a la ubicación de los mintérminos, como se advierte en la figura. Si se compara con la tabla de verdad, la ventaja del mapa de Karnaugh es que permite reconocer con facilidad los mintérminos que pueden combinarse aplicando la propiedad 14*a* de la sección 2.5. Los mintérminos en cualesquiera dos celdas adyacentes, ya sea en la misma fila o en la misma columna, pueden combinarse. Por ejemplo, los mintérminos m_2 y m_3 pueden combinarse como

$$\begin{aligned} m_2 + m_3 &= x_1\overline{x}_2 + x_1x_2 \\ &= x_1(\overline{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$

x_1	x_2	
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

a) Tabla de verdad



b) Mapa de Karnaugh

Figura 4.2 Ubicación de los mintérminos de dos variables.

El mapa de Karnaugh no sólo es útil para combinar pares de mintérminos. Como veremos en varios ejemplos más grandes, puede usarse directamente para derivar un circuito de costo mínimo de una función lógica.

Mapa de dos variables

En la figura 4.3 se ilustra un mapa de Karnaugh para una función de dos variables. Corresponde a la función f de la figura 2.15. El valor de f para cada combinación de las variables x_1 y x_2 se indica en la celda correspondiente del mapa. Como aparece 1 en ambas celdas de la fila inferior, las cuales son adyacentes, hay un solo término producto que puede hacer que f sea igual a 1 cuando las variables de entrada tienen los valores correspondientes a cualquiera de esas celdas. Para indicar este hecho, hemos encerrado en un círculo las entradas de las celdas en el mapa. En vez de aplicar la propiedad combinatoria de manera formal, es posible derivar el término producto intuitivamente. Ambas celdas se identifican mediante $x_2 = 1$, pero $x_1 = 0$ para la celda izquierda y $x_1 = 1$ para la celda derecha. Por tanto, si $x_2 = 1$, entonces $f = 1$ independientemente de que x_1 sea igual a 0 o a 1. El término producto que representa las dos celdas es simplemente x_2.

De manera similar, $f = 1$ para ambas celdas en la primera columna. Dichas celdas se identifican mediante $x_1 = 0$. En consecuencia, conducen al término producto $\bar{x}_1$. En virtud de que esto considera todos los casos en los que $f = 1$, se deduce que la realización de costo mínimo de la función es

$$f = x_2 + \bar{x}_1$$

Evidentemente, para encontrar una implementación de costo mínimo de una función es preciso hallar el número más pequeño de términos producto que producen un valor de 1 para todos

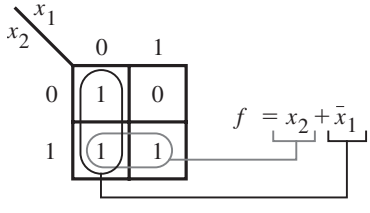


Figura 4.3 La función de la figura 2.15.

los casos donde $f = 1$. Más aún, el costo de estos términos producto debe ser lo más bajo posible. Nótese que es más barato implementar un término producto que abarca dos celdas adyacentes que un término que cubre una sola celda. Para nuestro ejemplo, una vez que el término producto x_2 cubre las dos celdas en la fila inferior, sólo queda una celda (la superior izquierda). Aunque podría abarcarse mediante el término $\overline{x}_1\overline{x}_2$, es mejor combinar las dos celdas de la columna izquierda para producir el término producto $\overline{x}_1$ porque es más barato implementarlo.

Mapa de tres variables

Un mapa de Karnaugh de tres variables se construye colocando dos mapas de dos variables lado a lado. En la figura 4.4 se muestra el mapa y se indican las ubicaciones de los mintérminos. En este caso cada combinación de x_1 y x_2 identifica una columna del mapa, mientras que el valor de x_3 distingue las dos filas. Para asegurar que los mintérminos de las celdas adyacentes del mapa siempre puedan combinarse en un solo término producto, tales celdas deben diferir en el valor de sólo una variable. Por tanto, las columnas se identifican mediante la sucesión de valores (x_1, x_2) de 00, 01, 11 y 10, en vez de los más obvios 00, 01, 10 y 11. Esto hace que la segunda y la tercera columnas sean diferentes sólo en la variable x_1. Además, la primera y la cuarta columnas difieren sólo en la variable x_1, lo que significa que esas columnas pueden considerarse adyacentes. El lector puede encontrar útil imaginar el mapa como un rectángulo plegado en un cilindro donde se tocan los bordes izquierdo y derecho de la figura 4.4*b*. (Una secuencia de códigos, o combinaciones, donde códigos consecutivos difieren sólo en una variable se conoce como *código Gray*, que tiene varios propósitos, algunos de los cuales se abordan más adelante.)

En la figura 4.5*a* se representa la función de la figura 2.18 en forma de mapa de Karnaugh. Para sintetizar esta función es necesario cubrir los cuatro 1 del mapa tan eficientemente como sea posible. No es difícil ver que para ello bastan dos términos producto. El primero cubre los 1 de la fila superior, los cuales están representados por el término $x_1\overline{x}_3$. El segundo término es $\overline{x}_2x_3$, que cubre los 1 de la fila inferior. Por ende, la función se implementa como

$$f = x_1\overline{x}_3 + \overline{x}_2x_3$$

que describe el circuito obtenido en la figura 2.19*a*.

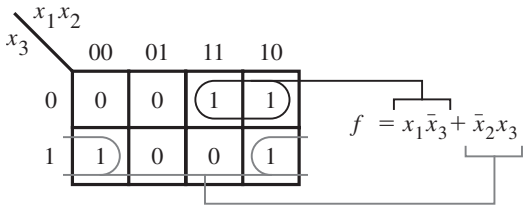
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

a) Tabla de verdad

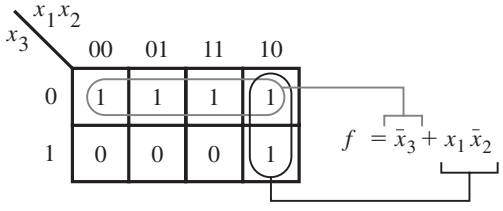
x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

b) Mapa de Karnaugh

Figura 4.4 Ubicación de mintérminos de tres variables.



a) La función de la figura 2.18



b) La función de la figura 4.1

Figura 4.5 Ejemplos de mapas de Karnaugh de tres variables.

En un mapa de tres variables es posible combinar celdas para producir términos producto que correspondan a una sola celda, a dos celdas adyacentes o a un grupo de cuatro celdas adyacentes. La realización de un grupo de cuatro celdas adyacentes que usan un solo término producto se ilustra en la figura 4.5*b,* usando la función de la figura 4.1. Las cuatro celdas de la fila superior corresponden a las combinaciones 000, 010, 110 y 100 de (x_1, x_2, x_3). Como ya explicamos, esto indica que si $x_3 = 0$, entonces $f = 1$ para las cuatro posibles combinaciones de x_1 y x_2, lo que significa que el único requisito es que $x_3 = 0$. Por ende, el término producto $\bar{x}_3$ representa esas cuatro celdas. El 1 restante, que corresponde al mintérmino m_5, se cubre mejor con el término $x_1\bar{x}_2$, que se obtiene al combinar las dos celdas de la columna de la derecha. La realización completa de f es

$$f = \bar{x}_3 + x_1\bar{x}_2$$

También es posible tener un grupo de ocho 1 en un mapa de tres variables. Éste es el caso trivial donde $f = 1$ para todas las combinaciones de variables de entrada; en otras palabras, f es igual a la constante 1.

El mapa de Karnaugh ofrece un mecanismo simple para generar los términos producto que han de usarse a fin de implementar una función. Un término producto debe incluir sólo las variables que tengan el mismo valor para todas las celdas en el grupo representado por ese término. Si la variable es igual a 1 en el grupo, aparece sin complementar en el término producto; si es igual a 0, aparece complementada. Cada variable que a veces es 1 y a veces 0 en el grupo no aparece en el término producto.

Mapa de cuatro variables

Un mapa de cuatro variables se construye colocando juntos dos mapas de tres variables para crear cuatro filas de la misma forma que empleamos dos mapas de dos variables para formar las cuatro columnas de un mapa de tres variables. En la figura 4.6 se muestra la estructura del mapa

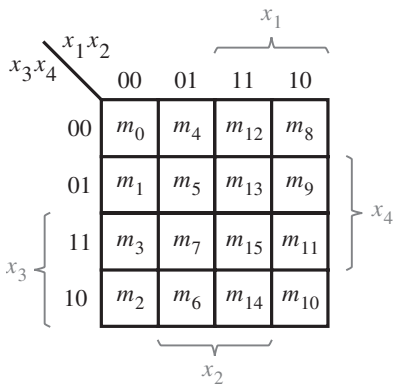


Figura 4.6 Mapa de Karnaugh de cuatro variables.

de cuatro variables y la ubicación de los mintérminos. En esta figura hemos incluido otra forma muy usada de diseñar las filas y las columnas. Como se muestra en gris, basta indicar las filas y las columnas para las que una variable es igual a 1. Por tanto, $x_1 = 1$ para las dos columnas de la extrema derecha, $x_2 = 1$ para las dos columnas de en medio, $x_3 = 1$ para las dos filas de la parte inferior y $x_4 = 1$ para las dos filas de en medio.

En la figura 4.7 se presentan cuatro ejemplos de funciones de cuatro variables. La función f_1 tiene un grupo de cuatro 1 en celdas adyacentes en las dos filas inferiores, para las cuales $x_2 = 1$ y $x_3 = 1$; se representan con el término producto $\overline{x}_2x_3$. Esto deja por cubrir los dos 1 de la segunda fila, lo que se logra con el término $x_1\overline{x}_3x_4$. En consecuencia, la implementación de costo mínimo de la función es

$$f_1 = \overline{x}_2x_3 + x_1\overline{x}_3x_4$$

La función f_2 incluye un grupo de ocho 1 que pueden implementarse mediante un solo término, x_3. De nuevo el lector debe notar que si los dos 1 restantes se implementasen por separado, el resultado sería el término producto $x_1\overline{x}_3x_4$. La implementación de estos 1 como parte de un grupo de cuatro 1, como se muestra en la figura, produce el término producto menos costoso x_1x_4.

Igual que los bordes izquierdo y derecho del mapa son adyacentes en términos de la asignación de las variables, los bordes superior e inferior también lo son. De hecho, las cuatro esquinas del mapa son adyacentes una a otra y, por tanto, pueden formar un grupo de cuatro 1, que puede implementarse con el término producto $\overline{x}_2\overline{x}_4$. Este caso queda descrito con la función f_3. Aparte de este grupo de 1, hay otros cuatro 1 que deben cubrirse para implementar f_3. Ello se logra como se muestra en la figura.

En todos los ejemplos considerados hasta el momento existe una solución única que conduce al circuito de costo mínimo. La función f_4 brinda un ejemplo en el que se tienen algunas opciones. Los grupos de cuatro 1 de las esquinas superior izquierda e inferior derecha del mapa se realizan mediante los términos $\overline{x}_1\overline{x}_3$ y x_1x_3, respectivamente. Esto deja los dos 1 correspondientes al término $x_1x_2\overline{x}_3$, pero éstos pueden realizarse de manera más económica si se tratan como parte de un grupo de cuatro 1. Pueden ser incluidos en dos diferentes grupos de cuatro,

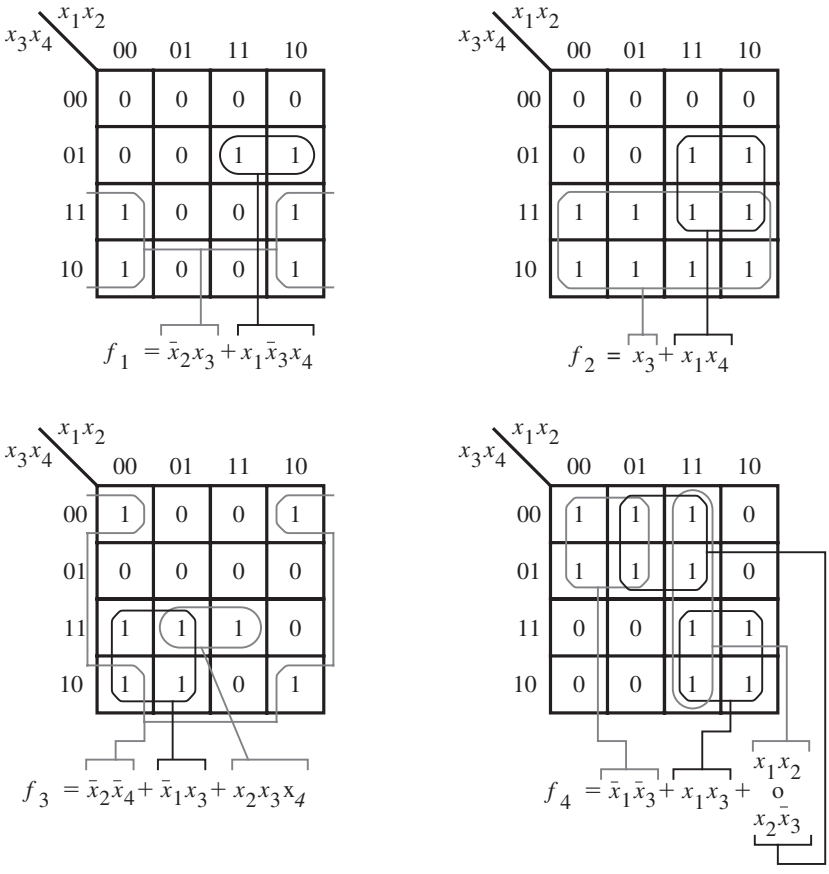


Figura 4.7 Ejemplos de mapas de Karnaugh de cuatro variables.

como se indica en la figura. Una posibilidad conduce al término producto x_1x_2 y la otra a $x_2\overline{x}_3$. Ambos términos tienen el mismo costo; por tanto, no importa cuál se elija en el circuito final. Nótese que el complemento de x_3 en el término $x_2\overline{x}_3$ no implica un aumento en el costo en comparación con x_1x_2, ya que este complemento debe generarse de cualquier forma para producir el término $\overline{x}_1\overline{x}_3$, el cual se incluye en la implementación.

Mapa de cinco variables

Pueden usarse dos mapas de cuatro variables para construir un mapa de cinco variables. Es fácil imaginar una estructura donde un mapa se halle directamente detrás de otro y ambos se distingan por $x_5 = 0$ para un mapa y $x_5 = 1$ para el otro. Como resulta complicado dibujar tal estructura, los dos mapas simplemente se colocan lado a lado, como se muestra en la figura 4.8. Para la función lógica dada en este ejemplo, dos grupos de cuatro 1 aparecen en el mismo lugar en los dos mapas de cuatro variables, en consecuencia su realización no depende del valor de x_5. Lo mismo es cierto para los dos grupos de dos 1 de la segunda fila. El 1 de la esquina superior

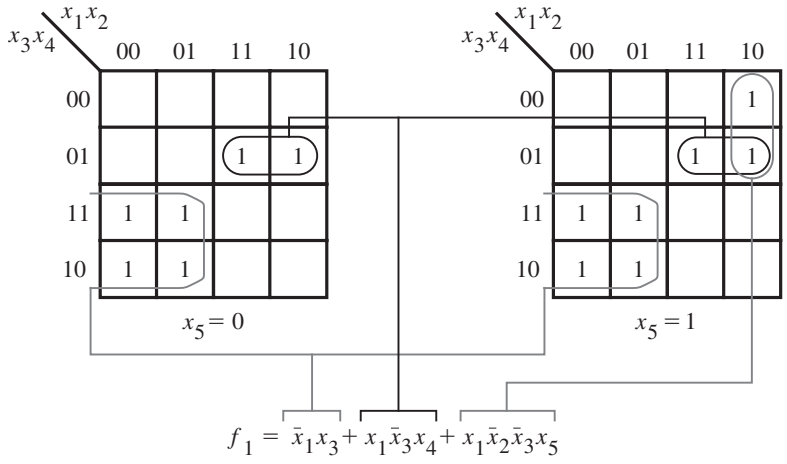


Figura 4.8 Mapa de Karnaugh de cinco variables.

derecha sólo aparece en el mapa de la derecha, donde $x_5 = 1$; forma parte del grupo de dos 1 realizado por el término $x_1\bar{x}_2\bar{x}_3x_5$. Nótese que en este mapa se dejan en blanco las celdas para las que $f = 0$ a fin de hacer que la figura sea más legible. Del mismo modo se hará en varios de los mapas que siguen.

El uso de un mapa de cinco variables es naturalmente más complicado que el de mapas con menos variables. La extensión del concepto de mapa de Karnaugh a más variables no resulta útil desde el punto de vista práctico. Esto no es problemático, puesto que la síntesis práctica de las funciones lógicas se realiza con herramientas CAD que llevan a cabo la minimización necesaria de forma automática. Aunque los mapas de Karnaugh ocasionalmente son útiles para diseñar circuitos lógicos pequeños, la razón principal de exponerlos es ofrecer un vehículo simple para ilustrar las ideas que el proceso de minimización supone.

4.2 ESTRATEGIA DE MINIMIZACIÓN

Para los ejemplos de la sección anterior recurrimos a un enfoque intuitivo a fin de decidir cómo han de agruparse los 1 en un mapa de Karnaugh para obtener la implementación de costo mínimo de una función. Nuestra estrategia intuitiva fue encontrar tan pocos y tan grandes grupos de 1 como fuera posible para cubrir todos los casos en que la función tuviese un valor de 1. Cada grupo de 1 debe abarcar celdas que puedan representarse mediante un solo término producto. Cuanto más grande sea el grupo de 1, menor será el número de variables en el término producto correspondiente. Este enfoque sirvió bien porque los mapas de Karnaugh de los ejemplos eran pequeños. Es inadecuado para funciones lógicas más grandes, con muchas variables. En vez de ello hay que contar con un método organizado para derivar una implementación de costo mínimo. En esta sección expondremos un posible método, similar a las técnicas automatizadas en

las herramientas CAD. Para ilustrar las ideas principales emplearemos mapas de Karnaugh. Más adelante, en la sección 4.8, describiremos otra forma de representar funciones lógicas, la cual se utiliza en las herramientas CAD.

4.2.1 TERMINOLOGÍA

En el desarrollo de técnicas para la síntesis de funciones lógicas se ha realizado una gran cantidad de trabajo de investigación. Sus resultados están publicados en varios documentos. En aras de facilitar la presentación de los resultados, ha evolucionado cierta terminología que evita usar frases muy descriptivas. En los párrafos siguientes definimos parte de esa terminología porque es útil para describir el proceso de minimización.

Literal

Un término producto consta de cierto número de variables, cada una de las cuales puede aparecer en forma complementada o sin complementar. Cada aparición de una variable, ya sea sin complementar o complementada, se llama *literal*. Por ejemplo, el término producto $x_1\overline{x}_2x_3$ tiene tres literales, y $\overline{x}_1x_3\overline{x}_4x_6$ cuatro.

Implicante

Un término producto que indica la combinación de entrada para la que una función es igual a 1 se llama *implicante* de la función. Los implicantes más básicos son los mintérminos, que explicamos en la sección 2.6.1. Para una función de n variables, un mintérmino es un implicante que consta de n literales.

Considérese la función de tres variables de la figura 4.9. Tiene 11 posibles implicantes, los cuales incluyen los cinco mintérminos: $\overline{x}_1\overline{x}_2\overline{x}_3$, $\overline{x}_1\overline{x}_2x_3$, $\overline{x}_1x_2\overline{x}_3$, $\overline{x}_1x_2x_3$ y $x_1x_2x_3$. Luego están los implicantes correspondientes a todos los posibles pares de mintérminos que pueden combinarse: $\overline{x}_1\overline{x}_2$($m_0$ y m_1), $\overline{x}_1\overline{x}_3$ (m_0 y m_2), $\overline{x}_1x_3$ (m_1 y m_3), $\overline{x}_1x_2$ (m_2 y m_3) y x_2x_3 (m_3 y m_7). Finalmente, existe un implicante que abarca un grupo de cuatro mintérminos, el cual consta de una sola literal $\overline{x}_1$.

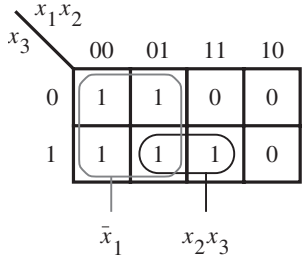


Figura 4.9 Función de tres variables $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

Implicante primo

Un implicante se llama *implicante primo* si no puede combinarse en otro implicante que tenga menos literales. Dicho de otra forma, es imposible borrar alguna literal en un implicante primo y aún así tener un implicante válido.

En la figura 4.9 hay dos implicantes primos: $\overline{x}_1$ y x_2x_3. No es posible borrar una literal en cualquiera de ellos. Hacerlo para $\overline{x}_1$ lo haría desaparecer. Para x_2x_3, borrar una literal dejaría x_2 o x_3. Pero x_2 no es un implicante porque incluye la combinación $(x_1, x_2, x_3) = 110$ para la que $f = 0$, y x_3 tampoco lo es porque incluye $(x_1, x_2, x_3) = 101$ para la que $f = 0$.

Cobertura

Un conjunto de implicantes que abarca todas las combinaciones para las que una función es igual a 1 se denomina *cobertura* de dicha función. Para la mayor parte de las funciones existen varias coberturas. Obviamente, un conjunto de todos los mintérminos para los que $f = 1$ es una cobertura. También es evidente que un conjunto de todos los implicantes primos es una cobertura.

Una cobertura define una implementación en particular de la función. En la figura 4.9 una cobertura que consta de mintérminos lleva a la expresión

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3$$

Otra cobertura válida está dada por la expresión

$$f = \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_2x_3$$

La cobertura que comprende los implicantes primos es

$$f = \overline{x}_1 + x_2x_3$$

Si bien todas estas expresiones representan la función f correctamente, la cobertura que consta de implicantes primos conduce a la implementación de menor costo.

Costo

En el capítulo 2 sugerimos que una buena indicación del costo de un circuito lógico es el número de compuertas en el circuito más el número total de entradas a todas ellas. A lo largo del libro usaremos esta definición de costo, pero supondremos que las entradas primarias, es decir, las variables de entrada, están disponibles en formas tanto verdadera como complementada sin ningún costo. Por tanto, la expresión

$$f = x_1\overline{x}_2 + x_3\overline{x}_4$$

tiene un costo de nueve porque puede implementarse con dos compuertas AND y una OR, con seis entradas a ellas.

Si dentro de un circuito se necesita una inversión, entonces la correspondiente compuerta NOT y su entrada se incluyen en el costo. Por ejemplo, la expresión

$$g = \overline{x_1\overline{x}_2 + x_3}(\overline{x}_4 + x_5)$$

se implementa con dos compuertas AND, dos OR y una NOT para complementar $(x_1\overline{x}_2 + x_3)$, con nueve entradas. Por ende, el costo total es 14.

4.2.2 PROCEDIMIENTO DE MINIMIZACIÓN

Hemos visto que es posible implementar una función lógica con varios circuitos, los cuales pueden tener diferentes estructuras y costos. Cuando se diseña un circuito lógico suele haber ciertos criterios que han de satisfacerse. Uno de ellos es el costo del circuito, tema que consideramos en la explicación previa. En general, cuanto más grande sea el circuito, mayor importancia tendrá el tema del costo. En esta sección supondremos que el objetivo central es obtener un circuito de costo mínimo.

Luego de decir que el costo es la preocupación principal, cabe señalar que otros criterios de optimación pueden ser más apropiados en ciertos casos. Por ejemplo, en el capítulo anterior describimos varios tipos de dispositivos lógicos programables (PLD) que tienen una estructura básica predefinida y pueden programarse para realizar varios circuitos. Para ellos el objeto principal radica en diseñar un circuito específico de modo que encaje en el dispositivo que se busca. Si ese circuito tiene o no el costo mínimo es algo irrelevante siempre que pueda realizarse bien en el dispositivo. Una herramienta CAD que tiene el propósito de diseñar con un dispositivo específico en mente desarrollará de manera automática optimaciones adecuadas para él. En la sección 4.6 mostraremos que la forma en la que debe optimarse un circuito puede ser diferente de acuerdo con el tipo de dispositivo.

En la subsección anterior llegamos a la conclusión de que la implementación de costo más bajo se logra cuando la cobertura de una función consta de implicantes primos. Cabe entonces preguntar cómo se determina el subconjunto de costo mínimo de implicantes primos que cubrirán la función. Es posible que algunos implicantes primos deban ser incluidos en la cobertura, mientras que para otros puede haber opciones. Si un implicante primo incluye un mintérmino para el que $f = 1$ que no está incluido en algún otro implicante primo, entonces se le debe incluir en la cobertura y se le llama *implicante primo esencial*. En el ejemplo de la figura 4.9, ambos implicantes primos son esenciales. El término x_2x_3 es el único implicante que cubre el mintérmino m_7, y $\overline{x}_1$ es el único que cubre los mintérminos m_0, m_1 y m_2. Nótese que el mintérmino m_3 se cubre con estos dos implicantes primos. La realización de costo mínimo de la función es

$$f = \overline{x}_1 + x_2x_3$$

Ahora presentaremos varios ejemplos en los que hay una opción respecto de qué implicantes primos incluir en la cobertura final. Considérese la función de cuatro variables de la figura 4.10. Hay cinco implicantes primos: $\overline{x}_1x_3$, $\overline{x}_2x_3$, $\underline{x}_3\overline{x}_4$, $\overline{x}_1x_2x_4$ y $x_2\overline{x}_3x_4$. Los esenciales (resaltados en gris) son $\overline{x}_2x_3$ (debido a m_{11}), $x_3\overline{x}_4$ (debido a m_{14}) y $x_2\overline{x}_3x_4$ (debido a m_{13}). Estos mintérminos deben incluirse en la cobertura. Estos tres implicantes primos cubren todos los mintérminos para los que $f = 1$, excepto m_7. Es claro que m_7 puede cubrirse mediante $\overline{x}_1x_3$ o bien por $\overline{x}_1x_2x_4$. Puesto que $\overline{x}_1x_3$ tiene el costo más bajo, se elige para la cobertura. En consecuencia, la realización de costo mínimo es

$$f = \overline{x}_2x_3 + x_3\overline{x}_4 + x_2\overline{x}_3x_4 + \overline{x}_1x_3$$

A partir de lo anterior el proceso de encontrar un circuito de costo mínimo comprende los pasos siguientes:

1. Generar todos los implicantes primos para la función f.
2. Encontrar el conjunto de implicantes primos esenciales.

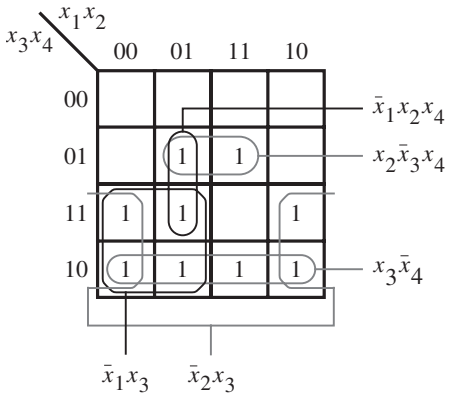


Figura 4.10 Función de cuatro variables $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

3. Si el conjunto de implicantes primos esenciales cubre todas las combinaciones para las que $f = 1$, entonces es la cobertura deseada de f. De otro modo hay que determinar los implicantes primos no esenciales que deben agregarse para formar una cobertura completa de costo mínimo.

La elección de los implicantes primos no esenciales que han de incluirse en la cobertura está regida por consideraciones de costo. Esta elección no siempre es obvia. De hecho, para funciones grandes podría haber muchas posibilidades y entonces hay que emplear un enfoque *heurístico* (es decir, que si bien considera sólo un subconjunto de posibilidades, brinda buenos resultados la mayor parte de las veces). Uno de tales enfoques consiste en elegir arbitrariamente un implicante primo no esencial, incluirlo en la cobertura y luego determinar el resto de ella. Después se determina otra cobertura con la suposición de que ese implicante no está en ella. Se comparan los costos de las coberturas resultantes y se elige implementar la menos costosa.

Podemos ilustrar el proceso con la función de la figura 4.11. De los seis implicantes primos, sólo $\bar{x}_3\bar{x}_4$ es esencial. Considérese a continuación $x_1x_2\bar{x}_3$ y supóngase primero que se incluirá en

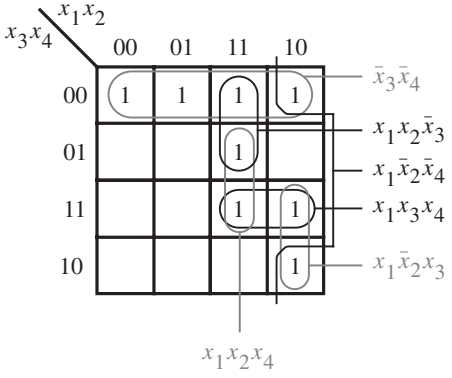


Figura 4.11 La función $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

la cobertura. Luego los restantes tres mintérminos, m_{10}, m_{11} y m_{15}, requerirán dos implicantes primos más para incluirse en la cobertura. Una posible implementación es

$$f = \bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_3x_4 + x_1\bar{x}_2x_3$$

La segunda posibilidad es que $x_1x_2\bar{x}_3$ no se incluya en la cobertura. Entonces $x_1x_2x_4$ se vuelve esencial porque no hay otra forma de cubrir m_{13}. Puesto que $x_1x_2x_4$ también cubre m_{15}, sólo m_{10} y m_{11} permanecen sin cubrir, lo que se logra con $x_1\bar{x}_2x_3$. Por tanto, la implementación alternativa es

$$f = \bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Es evidente que esta implementación es la mejor opción.

A veces puede no haber implicantes primos esenciales en absoluto. En la figura 4.12 se da un ejemplo. Elegir cualquiera de los implicantes primos, incluirlo y luego excluirlo de la cobertura lleva a dos posibilidades de igual costo. Una comprende los implicantes primos indicados en negro, lo que produce

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

La otra incluye los implicantes primos indicados en gris, lo que produce

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Este procedimiento puede aplicarse para encontrar las implementaciones de costo mínimo de funciones lógicas pequeñas o grandes. Para nuestros ejemplos pequeños fue conveniente usar mapas de Karnaugh a fin de determinar los implicantes primos de una función y luego elegir la cobertura final. En las herramientas CAD hay otras técnicas basadas en los mismos principios cuyo uso es más adecuado; las expondremos en las secciones 4.9 y 4.10.

Los ejemplos anteriores se basaron en la forma de suma de productos. A continuación ilustraremos los mismos conceptos para la forma de producto de sumas.

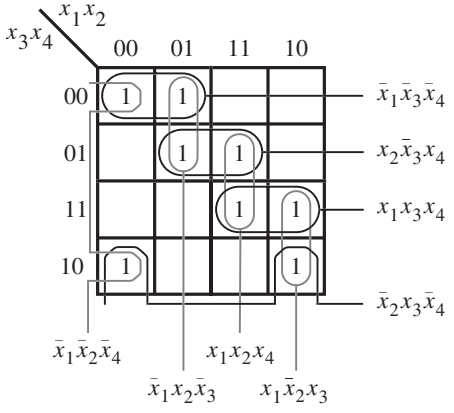


Figura 4.12 La función $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

4.3 MINIMIZACIÓN DE FORMAS DE PRODUCTO DE SUMAS

Ahora que sabemos cómo encontrar las implementaciones de funciones de costo mínimo en suma de productos (SOP, *sum-of-products*), podemos aplicar las mismas técnicas y el principio de dualidad para obtener implementaciones de costo mínimo en producto de sumas (POS, *product-of-sums*). En este caso son los maxitérminos para los que $f = 0$ los que deben combinarse en términos suma que sean lo más grande posible. De nuevo, un término suma se considera más grande si cubre más maxitérminos, y cuanto más grande el término, menos costosa su implementación.

En la figura 4.13 se muestra la misma función descrita en la figura 4.9. Hay tres maxitérminos que deben cubrirse: M_4, M_5 y M_6. Pueden cubrirse mediante los dos términos suma mostrados en la figura, lo que conduce a la implementación siguiente:

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

Un circuito correspondiente a esta expresión tiene dos compuertas OR y una AND, con dos entradas por cada una de ellas. Su costo es mayor que el de la implementación SOP equivalente derivada en la figura 4.9, que sólo requiere dos compuertas, una OR y una AND.

La función de la figura 4.10 se reproduce en la figura 4.14. Los maxitérminos para los que $f = 0$ pueden cubrirse como se muestra, llevan a la expresión

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

Esta expresión representa un circuito con tres compuertas OR y una AND. Dos de las compuertas OR tienen dos entradas, y la tercera cuatro; la compuerta AND tiene tres entradas. Si se supone que las versiones complementada y sin complementar de las variables de entrada x_1 a x_4 están disponibles sin costo adicional, el costo de este circuito es 15, lo cual resulta mejor que la implementación SOP derivada de la figura 4.10, que requiere cinco compuertas y 13 entradas a un costo total de 18.

En general, como ya aprendimos en la sección 2.6.1, las implementaciones SOP y POS de una función pueden no representar el mismo costo. Se alienta al lector a que halle las implementaciones POS para las funciones de las figuras 4.11 y 4.12, y compare los costos con las formas SOP.

Hemos mostrado cómo obtener implementaciones POS de costo mínimo encontrando los términos suma más grandes que cubran todos los maxitérminos para los que $f = 0$. Otra forma

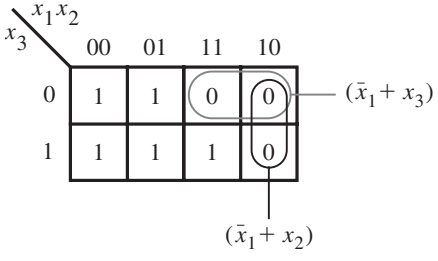


Figura 4.13 Minimización en POS de $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

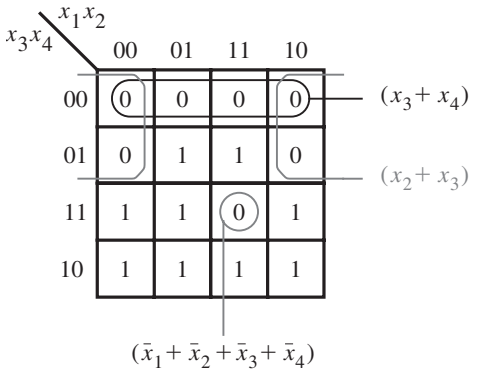


Figura 4.14 Minimización en POS de $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

de obtener el mismo resultado radica en hallar una implementación SOP de costo mínimo del complemento de f. Luego puede aplicarse el teorema de DeMorgan a esta expresión para obtener la realización POS más simple porque $f = \overline{\overline{f}}$. Por ejemplo, la implementación SOP más simple de $\overline{f}$ en la figura 4.13 es

$$\overline{f} = x_1\overline{x}_2 + x_1\overline{x}_3$$

Al complementar esta expresión mediante el teorema de DeMorgan se produce

$$\begin{aligned} f = \overline{\overline{f}} &= \overline{x_1\overline{x}_2 + x_1\overline{x}_3} \\ &= \overline{x_1\overline{x}_2} \cdot \overline{x_1\overline{x}_3} \\ &= (\overline{x}_1 + x_2)(\overline{x}_1 + x_3) \end{aligned}$$

que es el mismo resultado obtenido antes.

Con este enfoque, para la función de la figura 4.14 se obtiene

$$\overline{f} = \overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4$$

Si se complementa esta expresión se obtiene

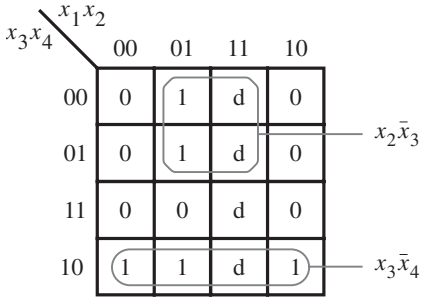
$$\begin{aligned} f = \overline{\overline{f}} &= \overline{\overline{x}_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\overline{x}_2\overline{x}_3} \cdot \overline{\overline{x}_3\overline{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4) \end{aligned}$$

que casa con la implementación antes derivada.

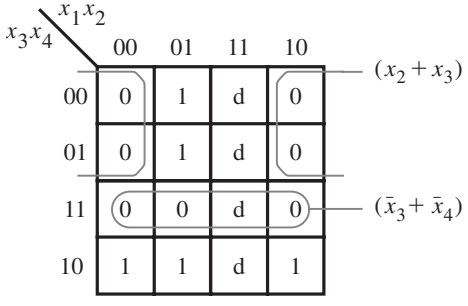
4.4 FUNCIONES ESPECIFICADAS DE MANERA INCOMPLETA

En los sistemas digitales suele ocurrir que ciertas condiciones de entrada nunca pueden suceder. Supóngase que x_1 y x_2 controlan dos interruptores interconectados de modo que ambos no pueden estar cerrados al mismo tiempo. Por tanto, los únicos tres estados posibles de los interruptores son que ambos estén abiertos o que uno esté abierto y el otro cerrado. Son posibles las combinaciones de entrada $(x_1, x_2) = 00$, 01 y 10, pero es seguro que 11 no ocurrirá. Entonces se dice que $(x_1, x_2) = 11$ es una *condición no-importa*, lo que significa que puede diseñarse un circuito con x_1 y x_2 como entradas e ignorar esta condición. Se dice que una función que tenga condiciones no-importa está *especificada de manera incompleta*.

Las condiciones no-importa, o simplemente los *no-importa*, sirven para sacar ventaja en el diseño de circuitos lógicos. Como tales combinaciones de entrada nunca ocurrirán, el diseñador puede suponer que el valor de la función para ellas es 1 o 0, lo que sea de mayor utilidad para encontrar una implementación de costo mínimo. En la figura 4.15 se ilustra esta idea. La función requerida tiene un valor de 1 para los mintérminos m_2, m_4, m_5, m_6 y m_{10}. Si se suponen los interruptores interconectados mencionados con antelación, las entradas x_1 y x_2 nunca serán iguales a 1 al mismo tiempo; por tanto, los mintérminos m_{12}, m_{13}, m_{14} y m_{15} se pueden usar como no-



a) Implementación SOP



b) Implementación POS

Figura 4.15 Dos implementaciones de la función $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

importa. Los no importa se denotan con la letra *d* en el mapa. En notación abreviada la función *f* se especifica como

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

donde *D* es el conjunto de no-importa.

En el inciso *a*) de la figura se indica la mejor implementación en suma de productos. Para formar los grupos más grandes posibles de 1 y, por ende, generar los implicantes primos de costo más bajo es preciso suponer que los no-importa D_{12}, D_{13} y D_{14} (que corresponden a los mintérminos m_{12}, m_{13} y m_{14}) tienen el valor de 1, mientras que D_{15} tiene el valor de 0. Entonces sólo existen dos implicantes primos, los cuales ofrecen una cobertura completa de *f*. La implementación resultante es

$$f = x_2\overline{x}_3 + x_3\overline{x}_4$$

En el inciso *b*) se indica cómo obtener la mejor implementación en producto de sumas. Los mismos valores se suponen para los no-importa. El resultado es

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)$$

La libertad para elegir el valor de los no-importa conduce a realizaciones muy simplificadas. Si inocentemente excluyéramos los no-importa de la síntesis de la función, al suponer que siempre tienen un valor de 0, la expresión SOP resultante sería

$$f = \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_3\overline{x}_4 + \overline{x}_2x_3\overline{x}_4$$

y la expresión POS sería

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2)$$

Ambas expresiones tienen costos mayores que las obtenidas con una asignación más apropiada de valores para los no-importa.

Aunque los valores no-importa pueden asignarse de manera arbitraria, ello podría no desembocar en una implementación de costo mínimo de una función. Si existen *k* no-importa, entonces hay 2^k posibles formas de asignarles los valores 0 o 1. En el mapa de Karnaugh puede verse cómo hacer mejor esta asignación para hallar la implementación más simple.

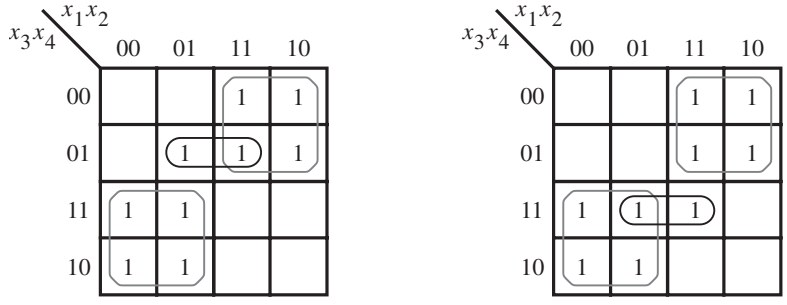
En el ejemplo anterior elegimos los no-importa D_{12}, D_{13} y D_{14} como iguales a 1 y D_{15} igual a 0 para ambas implementaciones, SOP y POS. Por ende, las expresiones derivadas representan la misma función, que también podría especificarse como $\sum m(2, 4, 5, 6, 10, 12, 13, 14)$. Asignar los mismos valores a los no-importa para ambas implementaciones, SOP y POS, no siempre es lo mejor. En ocasiones puede ser ventajoso dar a un no-importa en particular el valor 1 para la implementación SOP y el 0 para la POS, o viceversa. En tales casos, las expresiones óptimas SOP y POS representarán funciones diferentes, pero sólo diferirán por las combinaciones que correspondan a dichos no-importa. En el ejemplo 4.24 de la sección 4.14 se ilustra esta posibilidad.

El uso de interruptores interconectados para ilustrar cómo pueden ocurrir en un sistema real las condiciones no-importa puede parecer un tanto artificioso. Sin embargo, en los capítulos 6, 8 y 9 presentaremos numerosos ejemplos de no-importa que ocurren en el curso del diseño práctico de circuitos digitales.

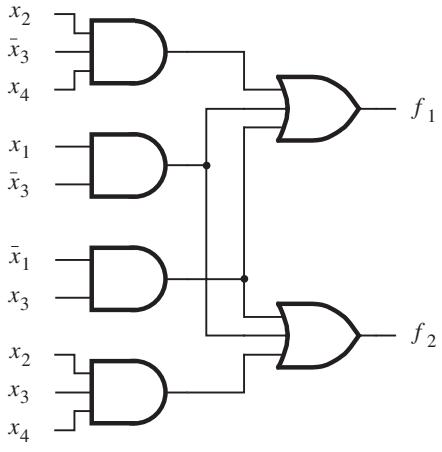
4.5 CIRCUITOS DE SALIDA MÚLTIPLE

En todos los ejemplos previos hemos considerado funciones solas y sus implementaciones en circuito. En los sistemas digitales prácticos es preciso implementar varias funciones como parte de algún circuito lógico grande. Los circuitos que implementan tales funciones suelen combinarse en un solo circuito menos costoso con varias salidas compartiendo algunas de las compuertas necesarias en la implementación de funciones individuales.

Ejemplo 4.1 En la figura 4.16 se ofrece un ejemplo de cómo se comparte una compuerta. Se deben implementar dos funciones, f_1 y f_2, de las mismas variables. Las implementaciones de costo mínimo



a) Función f_1 *b*) Función f_2



c) Circuito combinado para f_1 y f_2

Figura 4.16 Ejemplo de síntesis de salida múltiple.

para ellas se obtienen como se muestra en los incisos *a*) y *b*) de la figura. Esto resulta en las expresiones

$$f_1 = x_1\overline{x}_3 + \overline{x}_1x_3 + x_2\overline{x}_3x_4$$
$$f_2 = x_1\overline{x}_3 + \overline{x}_1x_3 + x_2x_3x_4$$

El costo de f_1 es cuatro compuertas y 10 entradas, para un total de 14. El costo de f_2 es el mismo. Por ende, el costo total es 28 si ambas funciones se implementan en circuitos separados. Si los dos circuitos se combinan en uno solo con dos salidas, es posible una realización menos costosa. Puesto que los dos primeros términos producto son idénticos en ambas expresiones, no se necesita duplicar las compuertas AND que las implementan. El circuito combinado se muestra en la figura 4.16*c*. Su costo es de seis compuertas y 16 entradas, para un total de 22.

En este ejemplo redujimos el costo global encontrando las realizaciones de costo mínimo de f_1 y f_2, y luego compartiendo las compuertas que implementan los términos producto comunes. Esta estrategia no siempre resulta la mejor, como se muestra en el ejemplo siguiente.

Ejemplo 4.2

En la figura 4.17 se presentan dos funciones que hay que implementar con un solo circuito. Las realizaciones de costo mínimo de las funciones individuales f_3 y f_4 se obtienen de los incisos *a*) y *b*) de la figura.

$$f_3 = \overline{x}_1x_4 + x_2x_4 + \overline{x}_1x_2x_3$$
$$f_4 = x_1x_4 + \overline{x}_2x_4 + \overline{x}_1x_2x_3\overline{x}_4$$

Ninguna de las compuertas AND puede compartirse, lo que significa que el costo del circuito combinado sería de seis compuertas AND, dos compuertas OR y 21 entradas, para un total de 29.

Pero varias otras realizaciones son posibles. En lugar de derivar las expresiones para f_3 y f_4 usando sólo implicantes primos, se buscan otros implicantes que puedan compartirse de ventajosamente en la realización combinada de las funciones. En la figura 4.17*c* se muestra la mejor elección de implicantes, lo que produce la realización

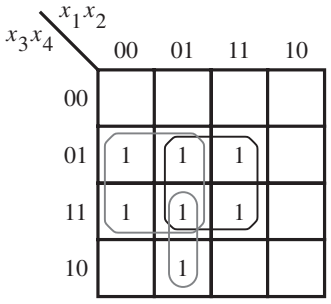
$$f_3 = x_1x_2x_4 + \overline{x}_1x_2x_3\overline{x}_4 + \overline{x}_1x_4$$
$$f_4 = x_1x_2x_4 + \overline{x}_1x_2x_3\overline{x}_4 + \overline{x}_2x_4$$

Los primeros dos implicantes son idénticos en ambas expresiones. El circuito resultante se muestra en la figura 4.17*d*. Tiene el costo de seis compuertas y 17 entradas, para un total de 23.

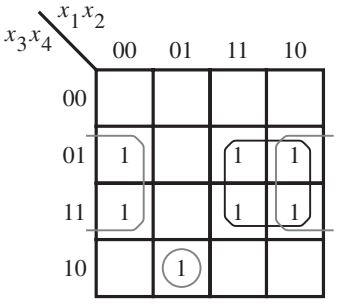
Ejemplo 4.3

En el ejemplo 4.1 se buscaba la mejor implementación en SOP para las funciones f_1 y f_2 de la figura 4.16. Ahora consideraremos la implementación en POS de las mismas funciones. Las expresiones en POS de costo mínimo para f_1 y f_2 son

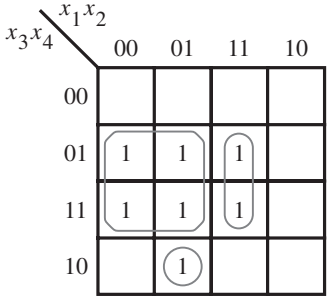
$$f_1 = (\overline{x}_1 + \overline{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3 + x_4)$$

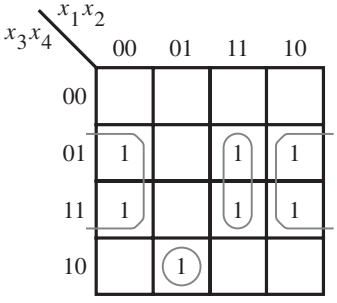


a) Realización óptima de f_3

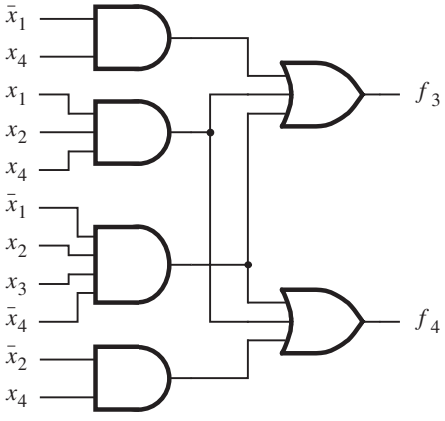


b) Realización óptima de f_4





c) Realización óptima de f_3 y f_4 juntas



d) Circuito combinado para f_3 y f_4

Figura 4.17 Otro ejemplo de síntesis de salida múltiple.

En estas expresiones no hay términos suma comunes que puedan compartirse en la implementación. Más aún, a partir de los mapas de Karnaugh de la figura 4.16 es claro que no hay término suma (que cubra las celdas donde $f_1 = f_2 = 0$) que pueda usarse provechosamente en la realización tanto de f_1 como de f_2. Por tanto, la mejor elección es implementar cada función por separado, de acuerdo con las expresiones precedentes. Cada función requiere tres compuertas OR, una AND y 11 entradas. En consecuencia, el costo total del circuito que implementa ambas funciones es 30. Esta realización es más costosa que la realización en SOP derivada en el ejemplo 4.1.

Ejemplo 4.4

Considere ahora la realización en POS de las funciones f_3 y f_4 de la figura 4.17. Las expresiones en POS de costo mínimo para f_3 y f_4 son

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(\overline{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + \overline{x}_4)$$

Los primeros tres términos suma son los mismos tanto en f_3 como en f_4; pueden compartirse en un circuito combinado. Esos términos requieren tres compuertas OR y seis entradas. Además, para f_3 se necesita una compuerta OR de dos entradas y una compuerta AND de cuatro entradas, y para f_4 se requiere una compuerta OR de tres entradas y una AND de cuatro. Por tanto, el circuito combinado comprende cinco compuertas OR, dos AND y 19 entradas, para un costo total de 26. Este costo es ligeramente superior que el del circuito derivado en el ejemplo 4.2.

Estos ejemplos muestran que las complejidades de las mejores implementaciones en SOP o POS de funciones concretas pueden ser muy diferentes. Para las funciones de las figuras 4.16 y 4.17 la forma SOP da mejores resultados. Pero si uno está interesado en la implementación de los complementos de las cuatro funciones de estas figuras, entonces la forma POS sería menos costosa.

Las más modernas herramientas CAD para sintetizar funciones lógicas automáticamente desarrollarán los tipos de optimaciones que se ilustran en los ejemplos precedentes.

4.6 SÍNTESIS MULTINIVEL

En las secciones anteriores nuestro objetivo consistió en encontrar una realización de costo mínimo, en suma de productos o en producto de sumas, de una función lógica. Los circuitos lógicos de este tipo tienen *dos niveles* (etapas) de compuertas. En la forma de suma de productos, el primer nivel comprende compuertas AND conectadas a una compuerta OR de segundo nivel. En la forma de producto de sumas, las compuertas OR de primer nivel alimentan la compuerta AND de segundo nivel. Hemos supuesto que ambas versiones de las variables de entrada, verdaderas y complementadas, están disponibles, de modo que no se necesitan compuertas NOT para complementar las variables.

Una realización de dos niveles es eficiente para funciones de unas cuantas variables. Sin embargo, conforme el número de entradas aumenta, un circuito de dos niveles puede resultar en

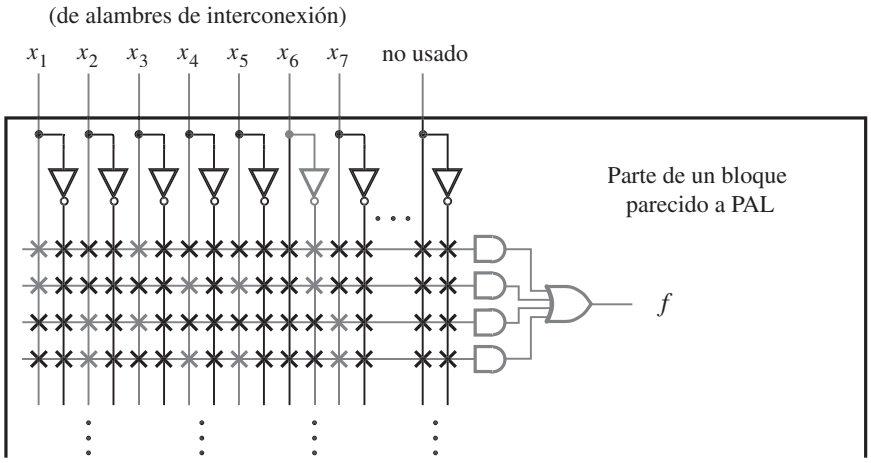


Figura 4.18 Implementación en un CPLD.

problemas de carga de entrada. Si éste es o no un conflicto depende del tipo de tecnología que se use para implementar el circuito. Considérese la función siguiente:

$$f(x_1, \ldots, x_7) = x_1x_3\overline{x}_6 + x_1x_4x_5\overline{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

Se trata de una expresión SOP de costo mínimo. Considérese ahora la implementación de *f* en dos tipos de PLD: un CPLD y un FPGA. En la figura 4.18 se muestra una parte de uno de los bloques parecidos a PAL de la figura 3.33. En la figura se indica en gris los circuitos utilizados para realizar la función *f*. Es claro que la forma SOP de la función es la adecuada para la arquitectura de chip del CPLD.

A continuación considérese la implementación de *f* en un FPGA. Para este ejemplo usaremos el FPGA presentado en la figura 3.39, que contiene dos entradas LUT. Como la expresión SOP para *f* requiere operaciones AND de tres y cuatro entradas, y una OR de cuatro, no puede implementarse directamente en este FPGA. El problema es que la carga de entrada requerida para implementar la función es muy alta para la arquitectura del chip objetivo.

Para resolver el problema de la carga de entrada, *f* debe expresarse en una forma que tenga más de dos niveles de operaciones lógicas. Tal forma se llama *expresión lógica multinivel*. Hay varios enfoques para sintetizar circuitos multinivel. Explicaremos dos importantes técnicas conocidas como *factorización* y *descomposición funcional*.

4.6.1 FACTORIZACIÓN

La propiedad distributiva presentada en la sección 2.5 permite factorizar la expresión anterior para *f* del modo siguiente

$$\begin{aligned} f &= x_1\overline{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5) \\ &= (x_1\overline{x}_6 + x_2x_7)(x_3 + x_4x_5) \end{aligned}$$

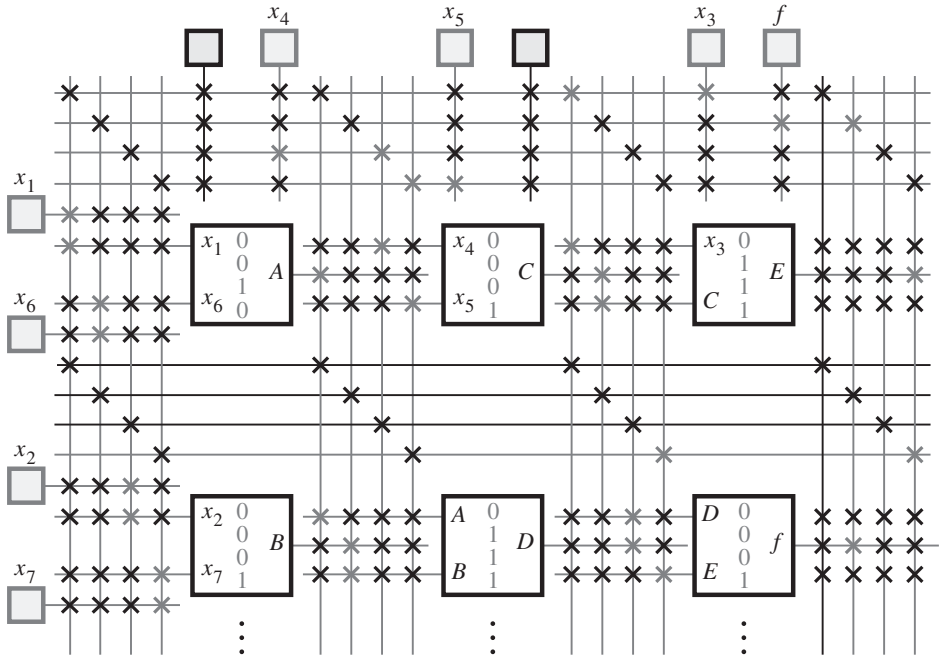


Figura 4.19 Implementación en un FPGA.

El circuito correspondiente tiene una carga de entrada máxima de dos; por tanto, se puede realizar usando LUT de dos entradas. En la figura 4.19 se brinda una posible implementación con el FPGA de la figura 3.39. Nótese que una función de dos variables que deba realizarse por cada LUT se indica en la caja que representa a la LUT.

Problema de carga de entrada

En el ejemplo anterior, las restricciones de carga de entrada fueron ocasionadas por la estructura fija del FPGA, donde cada LUT tiene sólo dos entradas. Sin embargo, aun cuando la arquitectura del chip objetivo no sea fija, la carga de entrada puede seguir siendo un problema. Para ilustrar esta situación, considérese la implementación de un circuito en un chip a la medida. Recuérdese que los chips a la medida contienen un gran número de compuertas. Si el chip se fabrica con tecnología CMOS, entonces habrá limitaciones de carga de entrada, como se expuso en la sección 3.8.8. En esta tecnología el número de entradas a una compuerta lógica debe ser pequeño. Por ejemplo, tal vez se quiera limitar la cantidad de entradas a una compuerta AND para que sean menos de cinco. Con esta restricción, si una expresión lógica incluye un término producto de siete entradas tendríamos que utilizar dos compuertas AND de cuatro entradas, como se indica en la figura 4.20.

Se puede usar factorización para enfrentar el problema de la carga de entrada. Supóngase de nuevo que las compuertas disponibles tienen una carga de entrada máxima de cuatro y que se quiere realizar la función

$$f = x_1\overline{x}_2x_3\overline{x}_4x_5x_6 + x_1x_2\overline{x}_3\overline{x}_4\overline{x}_5x_6$$

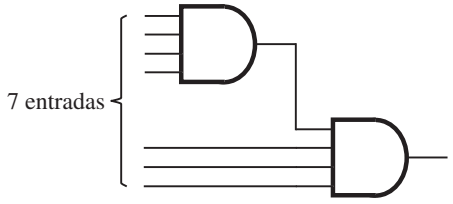


Figura 4.20 Uso de compuertas AND de cuatro entradas para realizar un término producto de siete entradas.

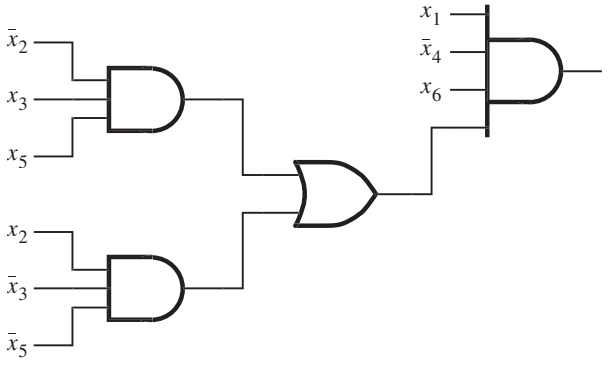


Figura 4.21 Circuito factorizado.

Ésta es una expresión mínima de suma de productos. Si empleamos el enfoque de la figura 4.20, se necesitarían cuatro compuertas AND y una OR para implementar esta expresión. Una mejor solución es factorizar la expresión como sigue

$$f = x_1\bar{x}_4x_6(\bar{x}_2x_3x_5 + x_2\bar{x}_3\bar{x}_5)$$

Entonces bastan tres compuertas AND y una OR para la realización de la función requerida, como se muestra en la figura 4.21.

Ejemplo 4.5 En situaciones prácticas, un diseñador de circuitos lógicos con frecuencia encuentra especificaciones que conducen naturalmente a un diseño inicial donde las expresiones lógicas están en forma factorizada. Suponga que se necesita un circuito que satisface los requisitos siguientes. Existen cuatro entradas: x_1, x_2, x_3 y x_4. Una salida, f_1, debe tener el valor 1 si al menos una de las entradas x_1 y x_2 es igual a 1 y tanto x_3 como x_4 son iguales a 1; también debe ser 1 si $x_1 = x_2 = 0$ y x_3 o x_4 es 1. En todos los demás casos $f_1 = 0$. Una salida diferente, f_2, será igual a 1 en todos los casos excepto cuando x_1 y x_2 sean iguales a 0 o cuando x_3 y x_4 sean iguales a 0.

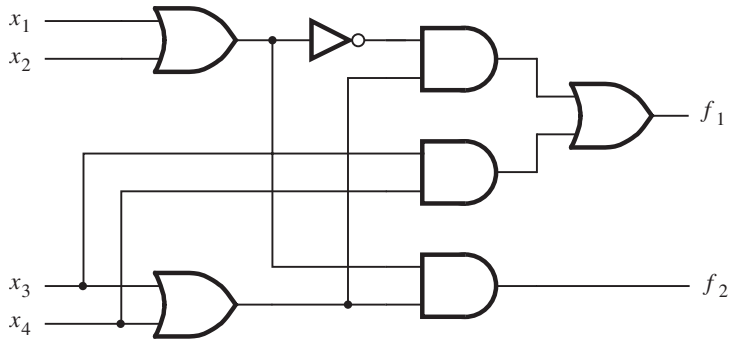


Figura 4.22 Circuito del ejemplo 4.5.

A partir de estas especificaciones, la función f_1 puede expresarse como

$$f_1 = (x_1 + x_2)x_3x_4 + \overline{x}_1\overline{x}_2(x_3 + x_4)$$

Esta expresión puede simplificarse en

$$f_1 = x_3x_4 + \overline{x}_1\overline{x}_2(x_3 + x_4)$$

que el lector puede comprobar con un mapa de Karnaugh.

La segunda función, f_2, se define con más facilidad en términos de su complemento, tal que

$$\overline{f}_2 = \overline{x}_1\overline{x}_2 + \overline{x}_3\overline{x}_4$$

Luego, el uso del teorema de DeMorgan produce

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

que es la expresión de costo mínimo para f_2; el costo aumenta de manera significativa si se usa la forma SOP.

Puesto que el objetivo es diseñar el circuito combinado de costo más bajo que implemente f_1 y f_2, parece que el mejor resultado se puede lograr si se usan las formas factorizadas para ambas funciones, caso en el que el término suma $(x_3 + x_4)$ puede compartirse. Más aún, al observar que $\overline{x}_1\overline{x}_2 = \overline{x_1 + x_2}$, el término suma $(x_1 + x_2)$ también puede compartirse si se expresa f_1 en la forma

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

Entonces el circuito combinado, que se muestra en la figura 4.22, comprende tres compuertas OR, tres AND, una NOT y 13 entradas, para un total de 20.

Impacto sobre la complejidad del cableado

El espacio de los chips de circuitos integrados está ocupado por los circuitos que implementan las compuertas lógicas y por los cables necesarios para hacer las conexiones entre ellas. La

cantidad de espacio necesario para cablear es una parte sustancial del área del chip. Por tanto, resulta útil mantener la complejidad del cableado cuan baja sea posible.

En una expresión lógica cada literal corresponde a un cable en el circuito que porta la señal lógica deseada. Puesto que la factorización reduce el número de literales, proporciona un poderoso mecanismo para reducir la complejidad del cableado de un circuito lógico. En el proceso de síntesis, las herramientas CAD consideran muchos aspectos, incluso el costo del circuito, la carga de entrada y la complejidad del cableado.

4.6.2 DESCOMPOSICIÓN FUNCIONAL

En los ejemplos anteriores, que ilustran el enfoque de factorización, se usaron circuitos multinivel para manejar las limitaciones de carga de entrada. Sin embargo, tales circuitos pueden ser preferibles a sus equivalentes de dos niveles aun si la carga de entrada no es un problema. En ciertos casos, los circuitos multinivel pueden reducir el costo de implementación. Por otra parte, casi siempre implican retrasos de propagación mayores porque utilizan múltiples etapas de compuertas lógicas. Analizaremos estos temas mediante algunos ejemplos.

En términos de cableado y compuertas lógicas, la complejidad de un circuito lógico a menudo puede reducirse por medio de la *descomposición* de un circuito de dos niveles en subcircuitos, donde uno o más de ellos implementan funciones que pueden usarse en varios lugares para construir el circuito final. Para lograr este objetivo, una expresión lógica de dos niveles se sustituye con dos o más expresiones nuevas, que entonces se combinan para definir un circuito multinivel. Podemos ilustrar esta idea con un ejemplo simple.

Ejemplo 4.6 Considere la expresión de costo mínimo en suma de productos

$$f = \overline{x}_1x_2x_3 + x_1\overline{x}_2x_3 + x_1x_2x_4 + \overline{x}_1\overline{x}_2x_4$$

y suponga que las entradas x_1 a x_4 sólo están disponibles en su forma verdadera. Entonces la expresión define un circuito con cuatro compuertas AND, una OR, dos NOT y 18 entradas (cables) a todas ellas. La carga de entrada es tres para las compuertas AND y cuatro para la OR. El lector debe observar que, en este caso, hemos incluido el costo de las compuertas NOT necesarias para complementar x_1 y x_2, en vez de suponer que ambas versiones, verdadera y complementada, de todas las variables de entrada estaban disponibles, como se hizo antes.

Al factorizar x_3 de los primeros dos términos y x_4 de los dos últimos esta expresión se convierte en

$$f = (\overline{x}_1x_2 + x_1\overline{x}_2)x_3 + (x_1x_2 + \overline{x}_1\overline{x}_2)x_4$$

Ahora sea $g(x_1, x_2) = \overline{x}_1x_2 + x_1\overline{x}_2$ y observe que

$$\begin{aligned}
\overline{g} &= \overline{\overline{x}_1x_2 + x_1\overline{x}_2} \\
&= \overline{\overline{x}_1x_2} \cdot \overline{x_1\overline{x}_2} \\
&= (x_1 + \overline{x}_2)(\overline{x}_1 + x_2) \\
&= x_1\overline{x}_1 + x_1x_2 + \overline{x}_2\overline{x}_1 + \overline{x}_2x_2 \\
&= 0 + x_1x_2 + \overline{x}_1\overline{x}_2 + 0 \\
&= x_1x_2 + \overline{x}_1\overline{x}_2
\end{aligned}$$

Entonces f puede escribirse como

$$f = gx_3 + \overline{g}x_4$$

que conduce al circuito mostrado en la figura 4.23, el cual requiere una compuerta OR adicional y una compuerta NOT para invertir el valor de g. Pero sólo necesita 15 entradas. Más aún, la carga de entrada más grande se redujo a dos. El costo de este circuito es menor que el de su equivalente de dos niveles. La consecuencia es un retraso de propagación mayor porque el circuito tiene tres niveles más de lógica.

En este ejemplo, la subfunción g es una función de las variables x_1 y x_2. Se utiliza como entrada al resto del circuito que completa la realización de la función requerida f. Sea h la función de esta parte del circuito, que depende sólo de tres entradas: g, x_3 y x_4. Luego la realización descompuesta de f puede expresarse algebraicamente como

$$f(x_1, x_2, x_3, x_4) = h[g(x_1, x_2), x_3, x_4]$$

La estructura de esta descomposición puede describirse en forma de diagrama de bloques como se muestra en la figura 4.24.

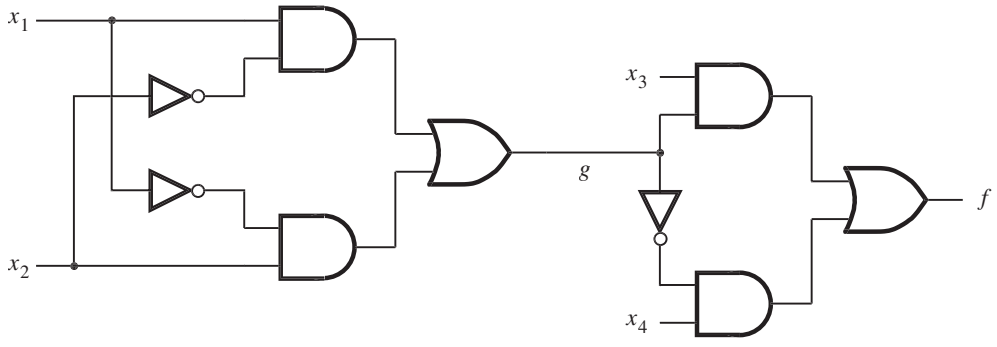


Figura 4.23 Circuito lógico del ejemplo 4.6.

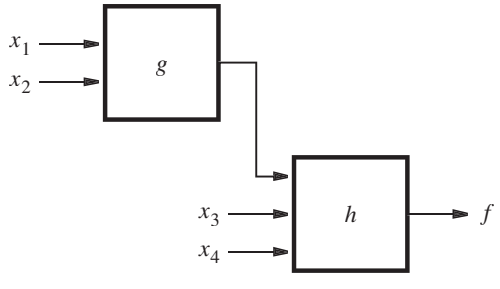
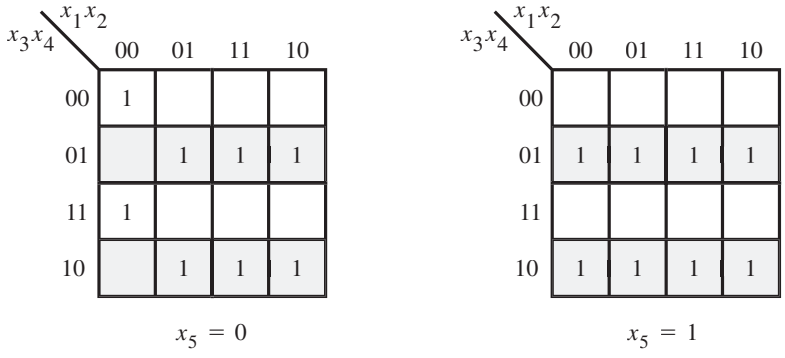


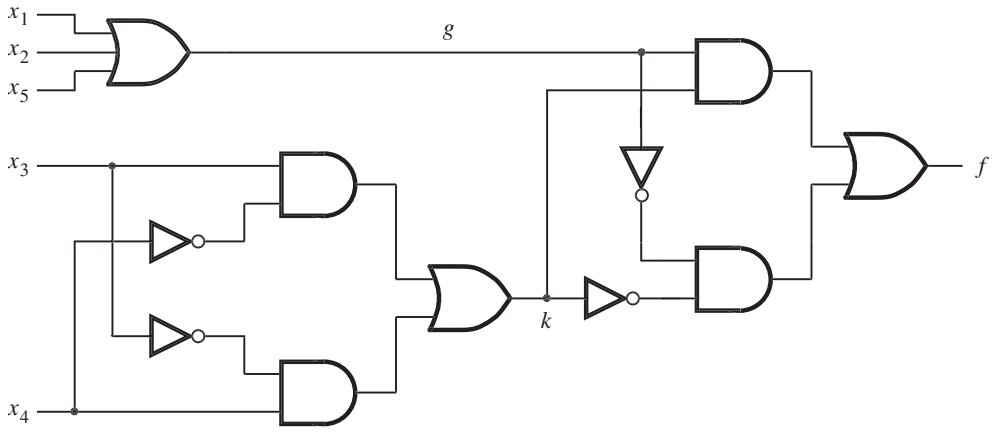
Figura 4.24 Estructura de la descomposición del ejemplo 4.6.

Aunque no es evidente a partir del primer ejemplo, la descomposición funcional puede conducir a mayores reducciones en la complejidad y el costo de los circuitos. El lector obtendrá un buen indicador de este beneficio a partir del ejemplo que sigue.

Ejemplo 4.7 En la figura 4.25*a* se define una función *f* de cinco variables en la forma de mapa de Karnaugh. Si se busca una buena descomposición para ella es necesario identificar primero las variables que se usarán como entradas a una subfunción. Una pista útil se obtiene a partir de los patrones de 1 en el mapa. Nótese que sólo existen dos patrones en las filas: la segunda y la cuarta tienen uno, resaltado en gris, mientras que la primera y la segunda tienen el otro. Una vez que se establece en qué fila se halla cada patrón, entonces el patrón mismo sólo depende de las variables que definen



a) Mapa de Karnaugh para la función f



b) Circuito obtenido al usar descomposición

Figura 4.25 Descomposición para el ejemplo 4.7.

las columnas en cada fila: x_1, x_2 y x_5. Sea $g(x_1, x_2, x_5)$ una subfunción que representa el patrón en las filas 2 y 4. Esta subfunción es justo

$$g = x_1 + x_2 + x_5$$

porque el patrón tiene un 1 donde cualquiera de dichas variables es igual a 1. Para especificar la ubicación de las filas donde ocurre el patrón g se utilizan las variables x_3 y x_4. Los términos $\overline{x}_3 x_4$ y $x_3 \overline{x}_4$ identifican la segunda y cuarta filas, respectivamente. Por ende, la expresión $(\overline{x}_3 x_4 + x_3 \overline{x}_4) \cdot g$ representa la parte de f definida en las filas 2 y 4.

A continuación, hay que encontrar una realización para el patrón de las filas 1 y 3, el cual tiene un 1 sólo en la celda donde $x_1 = x_2 = x_5 = 0$, que corresponde al término $\overline{x}_1 \overline{x}_2 \overline{x}_5$. Pero es posible hacer útil la observación de que este término es justo un complemento de g. La ubicación de las filas 1 y 3 se identifica por los términos $\overline{x}_3 \overline{x}_4$ y $x_3 x_4$, respectivamente. Por tanto, la expresión $(\overline{x}_3 \overline{x}_4 + x_3 x_4) \cdot \overline{g}$ representa f en las filas 1 y 3.

Se puede hacer otra útil observación. Las expresiones $(\overline{x}_3 x_4 + x_3 \overline{x}_4)$ y $(\overline{x}_3 \overline{x}_4 + x_3 x_4)$ son complementos una de la otra, como se muestra en el ejemplo 4.6. En consecuencia, si se hace $k(x_3, x_4) = \overline{x}_3 x_4 + x_3 \overline{x}_4$, la descomposición completa de f puede establecerse como

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h[g(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \overline{k}\,\overline{g} \end{aligned}$$

donde

$$\begin{aligned} g &= x_1 + x_2 + x_5 \\ k &= \overline{x}_3 x_4 + x_3 \overline{x}_4 \end{aligned}$$

El circuito resultante se muestra en la figura 4.25*b*. Requiere un total de 11 compuertas y 19 entradas. La carga de entrada más grande es tres.

En contraste, una expresión de costo mínimo en suma de productos para f es

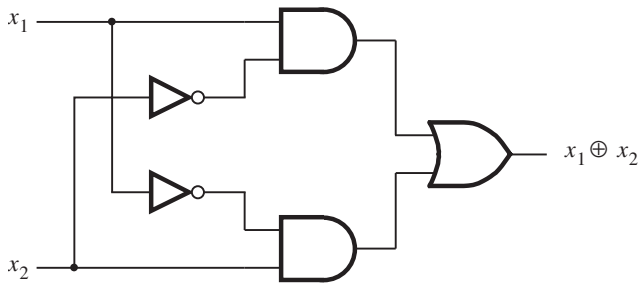
$$f = x_1 \overline{x}_3 x_4 + x_1 x_3 \overline{x}_4 + x_2 \overline{x}_3 x_4 + x_2 x_3 \overline{x}_4 + \overline{x}_3 x_4 x_5 + x_3 \overline{x}_4 x_5 + \overline{x}_1 \overline{x}_2 \overline{x}_3 \overline{x}_4 \overline{x}_5 + \overline{x}_1 \overline{x}_2 x_3 x_4 \overline{x}_5$$

El circuito correspondiente requiere un total de 14 compuertas (incluidas las cinco compuertas NOT para complementar las entradas primarias) y 41 entradas. La carga de entrada para la compuerta OR de salida es ocho. Obviamente, la descomposición funcional resulta en una implementación mucho más simple de esta función.

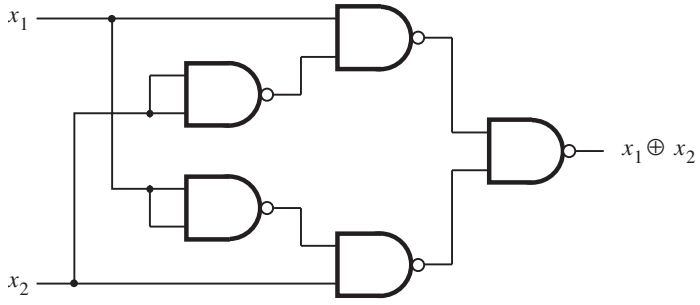
En los dos ejemplos anteriores, la descomposición es tal que una subfunción descompuesta depende de ciertas variables de entrada primarias, mientras que el resto de la implementación depende de las demás las variables. Tales descomposiciones reciben el nombre de *descomposiciones separadas* en los libros técnicos. Es posible tener una *descomposición no-separada*, donde las variables de la subfunción también se usan en la realización del resto del circuito. El ejemplo siguiente ilustra esta posibilidad.

Ejemplo 4.8

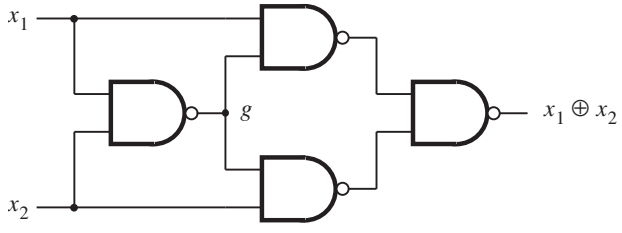
La OR exclusiva (XOR) es una función muy útil. En la sección 3.9.1 mostramos cómo puede realizarse con un circuito especial. También se puede realizar con compuertas AND y OR como



a) Implementación en suma de productos



b) Implementación con compuertas NAND



c) Implementación óptima con compuertas NAND

Figura 4.26 Implementación de XOR.

se muestra en la figura 4.26*a*. En la sección 2.7 explicamos cómo cualquier circuito AND-OR puede realizarse como un circuito NAND-NAND que tenga la misma estructura.

Ahora intentaremos explotar la descomposición funcional para encontrar una mejor implementación de XOR usando únicamente compuertas NAND. Sea el símbolo ↑ la representación de la operación NAND, de modo que $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$. Una expresión en suma de productos para la función XOR es

$$x_1 \oplus x_2 = x_1\overline{x}_2 + \overline{x}_1 x_2$$

Con base en el análisis de la sección 2.7, esta expresión puede escribirse en términos de operaciones NAND como

$$x_1 \oplus x_2 = (x_1 \uparrow \overline{x}_2) \uparrow (\overline{x}_1 \uparrow x_2)$$

Esta expresión requiere cinco compuertas NAND, y se implementa mediante el circuito de la figura 4.26*b*. Observe que un inversor se implementa con una compuerta NAND de dos entradas unidas.

Para hallar una descomposición puede manipularse el término $(x_1 \uparrow \overline{x}_2)$ del modo siguiente:

$$(x_1 \uparrow \overline{x}_2) = \overline{(x_1\overline{x}_2)} = \overline{(x_1(\overline{x}_1 + \overline{x}_2))} = (x_1 \uparrow (\overline{x}_1 + \overline{x}_2))$$

Es posible efectuar una manipulación similar de $(\overline{x}_1 \uparrow x_2)$ para generar

$$x_1 \oplus x_2 = (x_1 \uparrow (\overline{x}_1 + \overline{x}_2)) \uparrow ((\overline{x}_1 + \overline{x}_2) \uparrow x_2)$$

El teorema de DeMorgan afirma que $\overline{x}_1 + \overline{x}_2 = x_1 \uparrow x_2$; por tanto, puede escribirse

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Ahora se tiene una descomposición

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$

El circuito correspondiente, que sólo requiere cuatro compuertas NAND, se muestra en la figura 4.26*c*.

Aspectos prácticos

La descomposición funcional es una técnica poderosa para reducir la complejidad de los circuitos. También sirve para implementar funciones lógicas generales en circuitos que tienen restricciones inherentes. Por ejemplo, en los dispositivos lógicos programables (PLD), tema expuesto en el capítulo 3, es necesario "ajustar" el circuito lógico deseado en los bloques lógicos disponibles en dichos dispositivos. Los bloques disponibles son un blanco para subfunciones descompuestas que podrían emplearse para realizar funciones más grandes.

Un gran problema de la descomposición funcional consiste en encontrar las posibles subfunciones. Para funciones de muchas variables se debe probar un número enorme de posibilidades. Esta situación elimina los intentos por encontrar soluciones óptimas. En vez de ello se utilizan enfoques heurísticos que conducen a soluciones aceptables.

La explicación completa de la descomposición funcional y la factorización está más allá del ámbito de esta obra. El lector interesado puede consultar otras referencias [2-5]. Las modernas herramientas CAD usan mucho el concepto de descomposición.

4.6.3 CIRCUITOS NAND Y NOR MULTINIVEL

En la sección 2.7 mostramos que los circuitos de dos niveles que constan de compuertas AND y OR pueden convertirse fácilmente en circuitos que es posible realizar con compuertas NAND y NOR, usando el mismo arreglo de compuertas. En particular, un circuito AND-OR (suma de

productos) puede realizarse como un circuito NAND-NAND, mientras que un circuito OR-AND (producto de sumas) se convierte en un circuito NOR-NOR. Es posible usar el mismo enfoque de conversión para circuitos multinivel. Lo ilustramos con un ejemplo.

Ejemplo 4.9 En la figura 4.27*a* se presenta un circuito de cuatro niveles que consta de compuertas AND y OR. Primero derivemos un circuito funcionalmente equivalente que comprenda sólo compuertas NAND. Cada compuerta AND se convierte en una NAND al invertir su salida. Cada compuerta OR se convierte a una NAND al invertir sus entradas. Ésta es justo una aplicación del teorema de DeMorgan, como se ilustra en la figura 2.21*a*. En la figura 4.27*b* se muestran las inversiones necesarias en gris. Note que se aplica una inversión en ambos extremos de un cable dado. Ahora cada compuerta se convierte en una compuerta NAND. Esto explica el grueso de las inversiones agregadas al circuito original. Pero todavía hay cuatro inversiones que no forman parte de compuerta alguna; por tanto, deben implementarse por separado. Esas inversiones están en las entradas x_1, x_5, x_6 y x_7 y en la salida f. Pueden implementarse como dos compuertas NAND de dos entradas, donde las entradas están unidas. El circuito resultante se muestra en la figura 4.27*c*.

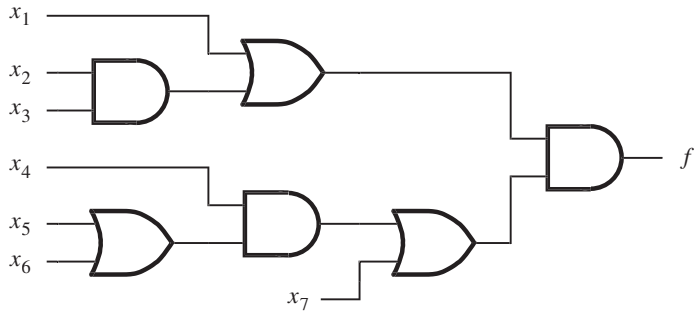
Es posible seguir un enfoque similar para convertir el circuito de la figura 4.27*a* en otro que comprenda sólo compuertas NOR. Una compuerta OR se convierte en una NOR invirtiendo su salida. Una AND se convierte en NOR si sus entradas se invierten, como se indica en la figura 2.21*b*. Con este enfoque, las inversiones necesarias para el circuito de nuestro ejemplo son las mostradas en gris en la figura 4.28*a*. Luego cada compuerta se convierte en una compuerta NOR. Las tres inversiones en las entradas x_2, x_3 y x_4 pueden realizarse como compuertas NOR de dos entradas, donde las entradas se unen. El circuito resultante se presenta en la figura 4.28*b*.

Es evidente que la topología básica de un circuito no cambia sustancialmente cuando se convierte de compuertas AND y OR en compuertas NAND o NOR. Sin embargo, puede ser necesario insertar compuertas adicionales para servir como compuertas NOT que implementen inversiones no absorbidas como parte de otras compuertas en el circuito.

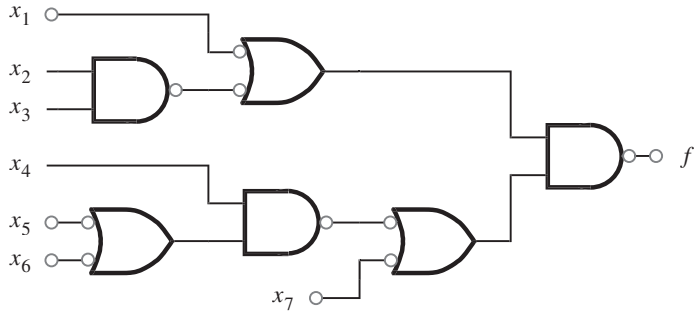
4.7 ANÁLISIS DE CIRCUITOS MULTINIVEL

En la sección precedente mostramos que puede resultar ventajoso implementar funciones lógicas mediante circuitos multinivel. También expusimos los enfoques más usados para sintetizar funciones de esta forma. En esta sección consideraremos la tarea de analizar un circuito para determinar la función que implementa.

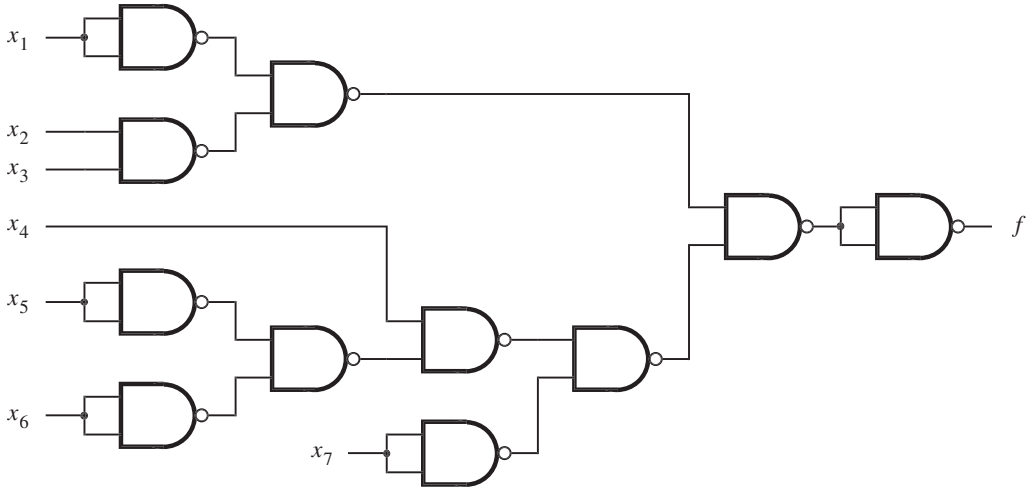
Para circuitos de dos niveles el proceso de análisis es simple. Si un circuito tiene una estructura AND-OR (NAND-NAND), entonces su función de salida puede escribirse en la forma SOP por inspección. De manera similar, es fácil derivar una expresión POS para un circuito OR-AND (NOR-NOR). La labor de análisis es más complicada para los circuitos multinivel porque es difícil escribir una expresión para la función por inspección. Hay que derivar la expresión deseada mediante el trazado del circuito y la determinación de su funcionalidad. El trazado puede realizarse primero desde el lado de entrada y luego hacia la salida, o primero en el lado de salida y después hacia atrás, a las entradas. En los puntos intermedios del circuito es preciso evaluar las subfunciones realizadas por las compuertas lógicas.



a) Circuito con compuertas AND y OR

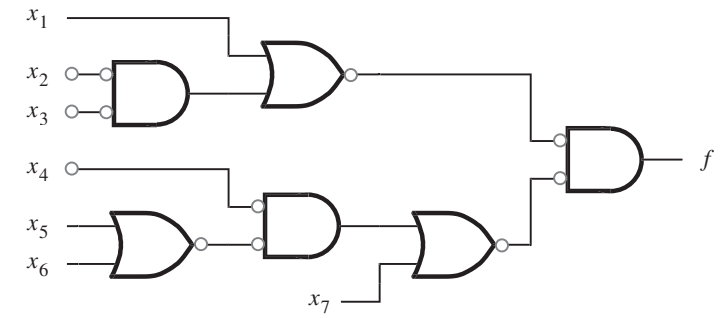


b) Inversiones necesarias para convertir a NAND

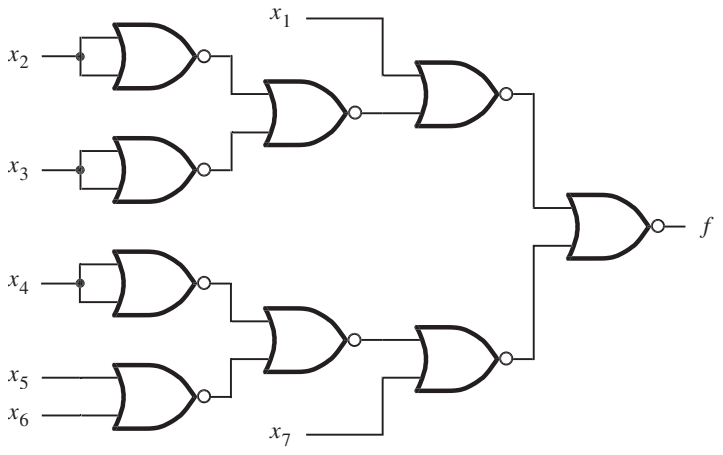


c) Circuito de compuerta NAND

Figura 4.27 Conversión a un circuito de compuerta NAND.



a) Inversiones necesarias para convertir a NOR



b) Circuito de compuerta NOR

Figura 4.28 Conversión a un circuito de compuerta NOR.

Ejemplo 4.10 En la figura 4.29 se copia el circuito de la figura 4.27*a*. Para determinar la función *f* que implementa puede considerarse la funcionalidad en puntos internos que son las salidas de varias compuertas. Tales puntos se etiquetan con P_1 a P_5 en la figura. Las funciones realizadas en ellos son

$$P_1 = x_2x_3$$
$$P_2 = x_5 + x_6$$
$$P_3 = x_1 + P_1 = x_1 + x_2x_3$$

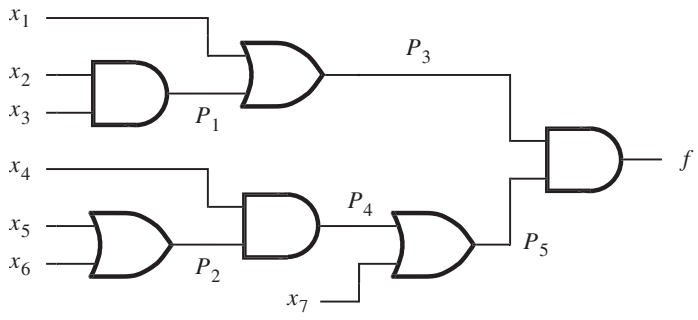


Figura 4.29 Circuito para el ejemplo 4.10.

$$P_4 = x_4P_2 = x_4(x_5 + x_6)$$
$$P_5 = P_4 + x_7 = x_4(x_5 + x_6) + x_7$$

Entonces f puede evaluarse como

$$\begin{aligned} f &= P_3P_5 \\ &= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7) \end{aligned}$$

Al aplicar la propiedad distributiva para eliminar los paréntesis se obtiene

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

Note que la expresión representa un circuito que comprende seis compuertas AND, una OR y 25 entradas. El costo de este circuito de dos niveles es mayor que el del circuito de la figura 4.29, pero tiene menor retardo de propagación.

Ejemplo 4.11

En el ejemplo 4.7 derivamos el circuito de la figura 4.25*b*. Además de las compuertas AND y OR, el circuito tiene algunas compuertas NOT. Se reproduce en la figura 4.30, y los puntos internos se etiquetan con P_1 a P_{10}, como se muestra. Ocurren las subfunciones siguientes

$$P_1 = x_1 + x_2 + x_5$$
$$P_2 = \overline{x}_4$$
$$P_3 = \overline{x}_3$$
$$P_4 = x_3P_2$$
$$P_5 = x_4P_3$$
$$P_6 = P_4 + P_5$$
$$P_7 = \overline{P}_1$$
$$P_8 = \overline{P}_6$$

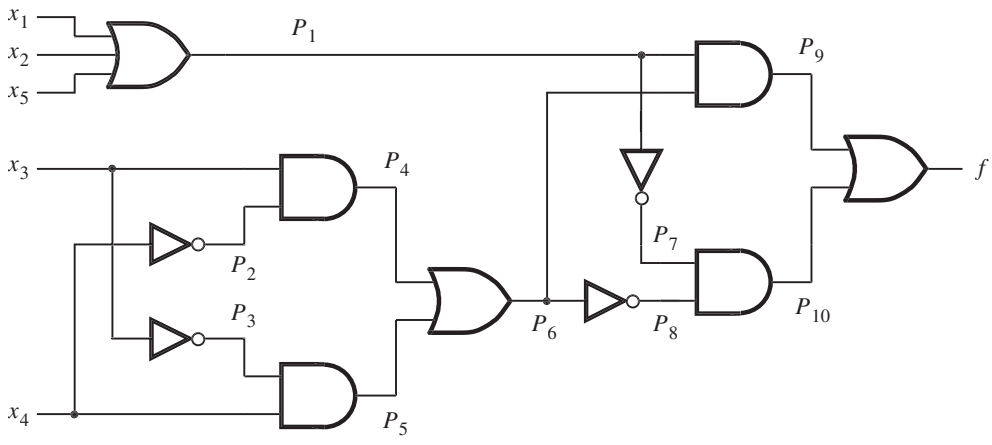


Figura 4.30 Circuito para el ejemplo 4.11.

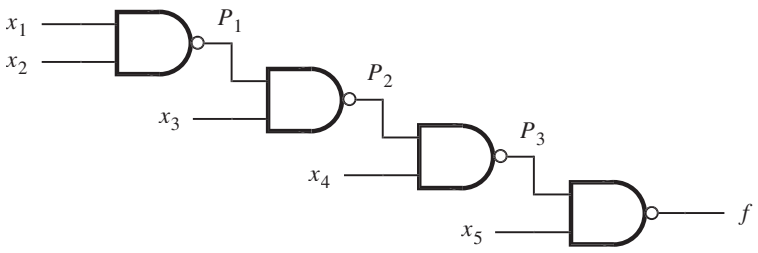
$$P_9 = P_1P_6$$
$$P_{10} = P_7P_8$$

Es posible derivar f trazando el circuito desde la salida hacia las entradas del modo siguiente

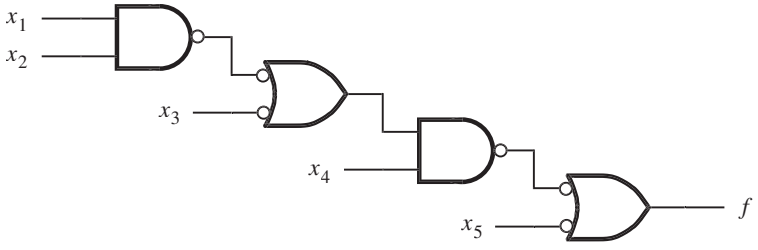
$$\begin{aligned}
f &= P_9 + P_{10} \\
&= P_1P_6 + P_7P_8 \\
&= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1\overline{P}_6 \\
&= (x_1 + x_2 + x_5)(x_3P_2 + x_4P_3) + \overline{x}_1\overline{x}_2\overline{x}_5\overline{P}_4\overline{P}_5 \\
&= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + x_4\overline{x}_3) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + \overline{P}_2)(\overline{x}_4 + \overline{P}_3) \\
&= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + \overline{x}_3x_4) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + x_4)(\overline{x}_4 + x_3) \\
&= x_1x_3\overline{x}_4 + x_1\overline{x}_3x_4 + x_2x_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_5x_3\overline{x}_4 + x_5\overline{x}_3x_4 + \\
&\quad \overline{x}_1\overline{x}_2\overline{x}_5\overline{x}_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_5x_4x_3
\end{aligned}$$

Ésta es la misma expresión que la presentada en el ejemplo 4.7.

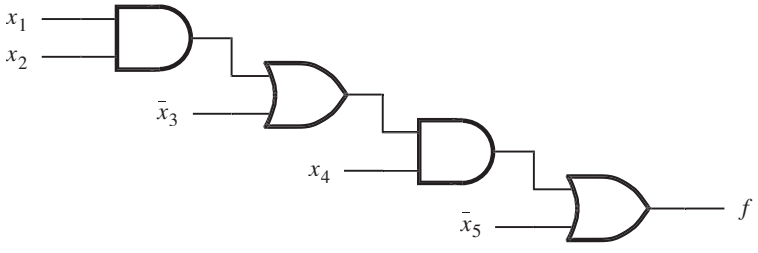
Ejemplo 4.12 Los circuitos que se basan en compuertas NAND y NOR son un poco más difíciles de analizar porque cada compuerta supone una inversión. En la figura 4.31*a* se describe un circuito de compuerta NAND simple que ilustra el efecto de las inversiones. Este circuito puede convertirse en uno con compuertas AND y OR invirtiendo el enfoque descrito en el ejemplo 4.9. Las burbujas que denotan inversiones pueden moverse, de acuerdo con el teorema de DeMorgan, como se indica en la figura 4.31*b*. Entonces el circuito puede convertirse en el presentado en el inciso



a) Circuito de compuertas NAND



b) Burbujas en movimiento para convertir a AND y OR



c) Circuito con compuertas AND y OR

Figura 4.31 Circuito para el ejemplo 4.12.

c) de ésta, el cual consta de compuertas AND y OR. Observe que, en el circuito convertido, las entradas x_3 y x_5 están complementadas. A partir de este circuito la función f se determina como

$$\begin{aligned} f &= (x_1x_2 + \overline{x}_3)x_4 + \overline{x}_5 \\ &= x_1x_2x_4 + \overline{x}_3x_4 + \overline{x}_5 \end{aligned}$$

No es necesario convertir un circuito NAND en uno con compuertas AND y OR para determinar su funcionalidad. Puede utilizarse el enfoque de los ejemplos 4.10 y 4.11 para derivar

f del modo siguiente. Sean P_1, P_2 y P_3 las etiquetas de los puntos internos, como se muestra en la figura 4.31*a*. Entonces

$$\begin{aligned}
P_1 &= \overline{x_1 x_2} \\
P_2 &= \overline{P_1 x_3} \\
P_3 &= \overline{P_2 x_4} \\
f &= \overline{P_3 x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2 x_4}} + \overline{x}_5 = P_2 x_4 + \overline{x}_5 \\
&= \overline{P_1 x_3} x_4 + \overline{x}_5 = (\overline{P}_1 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1 x_2}} + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (x_1 x_2 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= x_1 x_2 x_4 + \overline{x}_3 x_4 + \overline{x}_5
\end{aligned}$$

Ejemplo 4.13 El circuito de la figura 4.32 consta de compuertas NAND y NOR. Se puede analizar del modo siguiente

$$\begin{aligned}
P_1 &= \overline{x_2 x_3} \\
P_2 &= \overline{x_1 P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3 x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4 \overline{x}_5 \\
&= \overline{\overline{P_2 + P_3}} \cdot \overline{x}_5
\end{aligned}$$

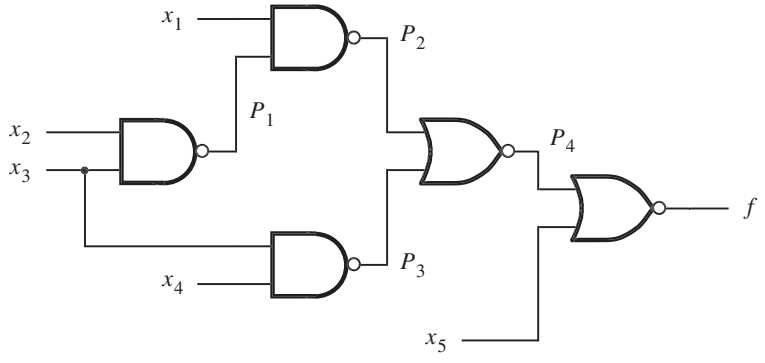


Figura 4.32 Circuito para el ejemplo 4.13.

$$= (P_2 + P_3)\overline{x}_5$$
$$= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4)\overline{x}_5$$
$$= (\overline{x}_1 + x_2x_3 + \overline{x}_3 + \overline{x}_4)\overline{x}_5$$
$$= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4)\overline{x}_5$$
$$= \overline{x}_1\overline{x}_5 + x_2\overline{x}_5 + \overline{x}_3\overline{x}_5 + \overline{x}_4\overline{x}_5$$

Note que al derivar de la segunda a la última línea se aplicó la propiedad 16*a* de la sección 2.5 para simplificar $x_2x_3 + \overline{x}_3$ en $x_2 + \overline{x}_3$.

El análisis de circuitos es mucho más simple que la síntesis. Con un poco de práctica es posible desarrollar la habilidad de analizar fácilmente incluso circuitos muy complejos.

Hasta ahora hemos cubierto una cantidad considerable de material respecto a la síntesis y el análisis de funciones lógicas. Hemos usado el mapa de Karnaugh como vehículo para ilustrar los conceptos implícitos en la búsqueda de las implementaciones óptimas de las funciones lógicas. También demostramos que las funciones lógicas pueden realizarse en varias formas, tanto con dos niveles de lógica como con niveles múltiples. En un entorno de diseño moderno, los circuitos lógicos se sintetizan con las herramientas CAD, no a mano. Los conceptos expuestos en el capítulo son muy generales; son representativos de las estrategias aplicadas en los algoritmos CAD. Como ya dijimos, no resulta apropiado emplear el mapa de Karnaugh para representar funciones lógicas en herramientas CAD. En la sección siguiente estudiaremos una representación alternativa de las funciones lógicas, adecuada para usarse en algoritmos CAD.

4.8 REPRESENTACIÓN CÚBICA

El mapa de Karnaugh es un excelente vehículo para ilustrar conceptos, e incluso es útil para diseño manual si las funciones sólo tienen unas cuantas variables. Para manejar funciones más grandes es necesario contar con técnicas algebraicas en vez de gráficas, las cuales pueden aplicarse a funciones de cualquier número de variables.

Se han desarrollado numerosas técnicas de optimación algebraica. No las explicaremos con gran detalle, sino que intentaremos ofrecer al lector una apreciación de las tareas involucradas. Esto ayuda a comprender mejor lo que las herramientas CAD pueden hacer y qué resultados cabe esperar de ellas. Los enfoques que expondremos utilizan una representación cúbica de las funciones lógicas.

4.8.1 CUBOS E HIPERCUBOS

Hasta el momento hemos visto cuatro formas de representar las funciones lógicas: tablas de verdad, expresiones algebraicas, diagramas de Venn y mapas de Karnaugh. Otra posibilidad es mapear una función de *n* variables en un cubo de *n* dimensiones.

Cubo bidimensional

En la figura 4.33 se muestra un cubo bidimensional. Las cuatro esquinas del cubo se llaman *vértices* y corresponden a las cuatro filas de una tabla de verdad. Cada vértice se identifica mediante dos coordenadas. Se supone que la coordenada horizontal corresponde a la variable x_1 y la vertical a x_2. Por tanto, el vértice 00 es la esquina inferior izquierda, que corresponde a la fila 0 en la tabla de verdad. El vértice 01 es la esquina superior izquierda, donde $x_1 = 0$ y $x_2 = 1$, que corresponde a la fila 1 en la tabla de verdad, y así por el estilo para los otros dos vértices.

Una función se mapeará en el cubo indicando con círculos gris los vértices para los que $f = 1$. En la figura 4.33, $f = 1$ para los vértices 01, 10 y 11. La función puede expresarse como un conjunto de vértices mediante la notación $f = \{01, 10, 11\}$. En la figura, la función f también se muestra en la forma de la tabla de verdad.

Un borde une dos vértices cuyas etiquetas sólo difieren en el valor de una variable. Por tanto, si dos vértices para los que $f = 1$ se unen mediante un borde, entonces éste representa la parte de la función igual que los dos vértices individuales. Por ejemplo, $f = 1$ para los vértices 10 y 11, que se unen mediante el borde etiquetado con 1x. Es costumbre utilizar la letra x para denotar el hecho de que la variable correspondiente puede ser 0 o 1. Por consiguiente, 1x significa que $x_1 = 1$, mientras que x_2 puede ser 0 o 1. De manera similar, los vértices 01 y 11 se unen mediante el borde etiquetado con x1, lo que indica que x_1 puede ser 0 o 1, pero $x_2 = 1$. El lector no debe confundir el empleo de la letra x para este propósito, en contraste con el uso donde x_1 y x_2 se refieren a las variables.

Dos vértices que se representan mediante un solo borde es la encarnación de la propiedad combinatoria 14*a* expuesta en la sección 2.5. El borde 1x es la suma lógica de los vértices 10 y 11. En esencia define el término x_1, que es la suma de los mintérminos $x_1\overline{x}_2$ y x_1x_2. La propiedad 14*a* indica que

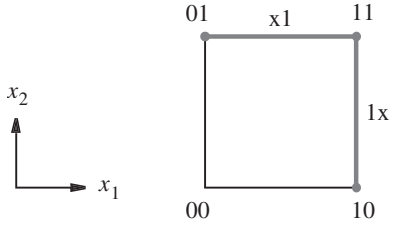
$$x_1\overline{x}_2 + x_1x_2 = x_1$$

En consecuencia, encontrar los bordes para los que $f = 1$ equivale a aplicar la propiedad combinatoria. Desde luego, esto también es semejante a hallar pares de celdas adyacentes en un mapa de Karnaugh para los que $f = 1$.

Los bordes 1x y x1 definen por completo la función de la figura 4.33; por tanto, la función puede representarse como $f = \{1x, x1\}$. Esto corresponde a la expresión lógica

$$f = x_1 + x_2$$

que también es obvia a partir de la tabla de verdad de la figura.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Figura 4.33 Representación de $f(x_1, x_2) = \sum m(1, 2, 3)$.

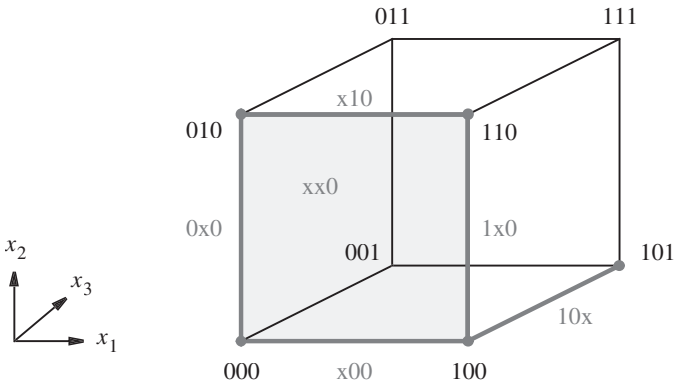


Figura 4.34 Representación de $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

Cubo tridimensional

En la figura 4.34 se ilustra un cubo tridimensional. Las coordenadas x_1, x_2 y x_3 son como se muestra a la izquierda. Cada vértice se identifica mediante una combinación específica de las tres variables. La función *f* mapeada en el cubo es la de la figura 4.1, que se usó en la figura 4.5*b*. Hay cinco vértices para los que $f = 1$: 000, 010, 100, 101 y 110. Estos vértices se unen mediante los cinco bordes que se muestran en gris: x00, 0x0, x10, 1x0 y 10x. Puesto que los vértices 000, 010, 100 y 110 incluyen todas las combinaciones de x_1 y x_2 cuando x_3 es 0, pueden especificarse mediante el término xx0. Este término significa que $f = 1$ si $x_3 = 0$, independientemente de los valores de x_1 y x_2. Nótese que xx0 representa el lado frontal del cubo, que está sombreado en gris.

Con base en la explicación anterior, es evidente que la función *f* puede representarse de varias formas, algunas de las cuales son

$$\begin{aligned} f &= \{000, 010, 100, 101, 110\} \\ &= \{0x0, 1x0, 101\} \\ &= \{x00, x10, 101\} \\ &= \{x00, x10, 10x\} \\ &= \{xx0, 10x\} \end{aligned}$$

En una realización física cada uno de los términos anteriores es un término producto implementado mediante una compuerta AND. Obviamente, el circuito menos costoso se obtiene si $f = \{xx0, 10x\}$, que equivale a la expresión lógica

$$f = \overline{x}_3 + x_1\overline{x}_2$$

Ésta es la expresión que derivamos usando el mapa de Karnaugh de la figura 4.5*b*.

Cubo de cuatro dimensiones

Las imágenes gráficas de los cubos bidimensionales y tridimensionales son fáciles de trazar. Un cubo de cuatro dimensiones es más difícil. Consta de dos cubos tridimensionales con

sus esquinas conectadas. La forma más simple de visualizar un cubo de cuatro dimensiones es colocando un cubo dentro de otro, como se muestra en la figura 4.35. Hemos supuesto que las coordenadas x_1, x_2 y x_3 son las mismas que en la figura 4.34, mientras que $x_4 = 0$ define el cubo exterior y $x_4 = 1$ el interior. La figura 4.35 indica cómo se mapea la función f_3 de la figura 4.7 en el cubo de cuatro dimensiones. Para no abarrotar la figura con demasiadas etiquetas, sólo hemos etiquetado los vértices para los que $f_3 = 1$. De nuevo se resaltan en gris todos los bordes que conectan dichos vértices.

Hay dos grupos de cuatro vértices adyacentes para los que $f_3 = 1$, los cuales pueden representarse como planos. El grupo que comprende 0000, 0010, 1000 y 1010 se representa mediante x0x0. El grupo 0010, 0011, 0110 y 0111 se representa con 0x1x. En la figura se sombrean esos planos. La función f_3 puede representarse de varias formas, por ejemplo

$$\begin{aligned} f_3 &= \{0000, 0010, 0011, 0110, 0111, 1000, 1010, 1111\} \\ &= \{00x0, 10x0, 0x10, 0x11, x111\} \\ &= \{x0x0, 0x1x, x111\} \end{aligned}$$

Puesto que cada x indica que es posible ignorar la variable correspondiente, ya que puede ser 0 o 1, el circuito más simple se obtiene si $f = \{x0x0, 0x1x, x111\}$, que equivale a

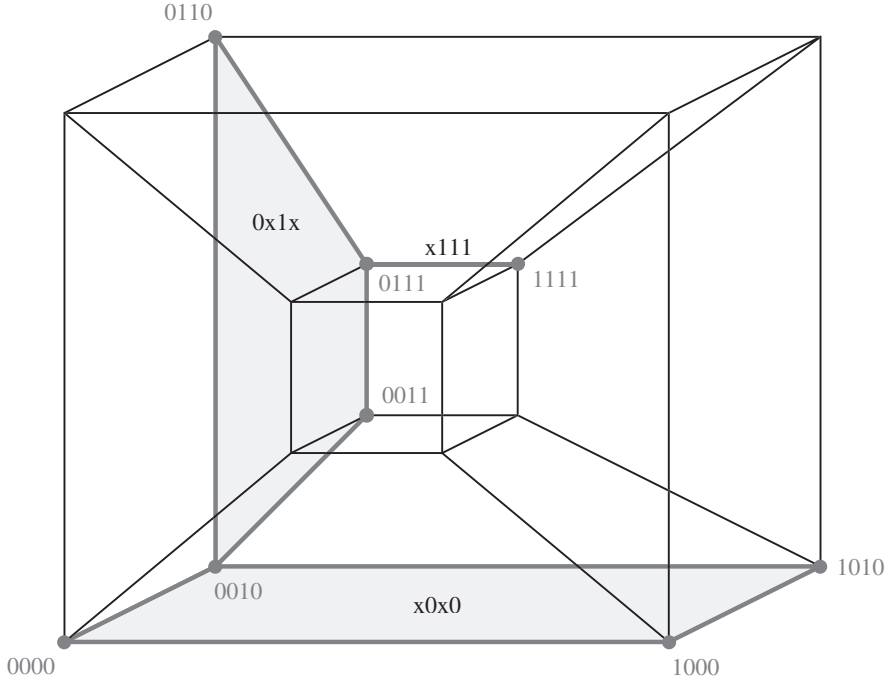


Figura 4.35 Representación de la función f_3 de la figura 4.7.

la expresión

$$f_3 = \overline{x}_2\overline{x}_4 + \overline{x}_1x_3 + x_2x_3x_4$$

La misma expresión se derivó en la figura 4.7.

Cubo de *n* dimensiones

Una función con *n* variables puede mapearse en un cubo de *n* dimensiones. Aunque resulta impráctico trazar imágenes de cubos que tengan más de cuatro variables, no es difícil extender las ideas presentadas líneas arriba a un caso general de *n* variables. Puesto que la interpretación visual no es posible y que normalmente la palabra *cubo* se usa sólo para una estructura tridimensional, muchas personas emplean la palabra *hipercubo* para referirse a estructuras con más de tres dimensiones. En este texto seguiremos empleando la palabra *cubo*.

Es conveniente referirse a un cubo cuyo *tamaño* refleje su número de vértices. Los vértices tienen el tamaño más pequeño. Cada variable tiene un valor de 0 o 1 en un vértice. Un cubo que tenga una x en la posición de una variable es más grande porque consta de dos vértices. Por ejemplo, el cubo 1x01 consta de los vértices 1001 y 1101. Un cubo que tenga dos x consta de cuatro vértices, y así sucesivamente. Un cubo que tenga *k* x consta de 2^k vértices.

Un cubo de *n* dimensiones tiene 2^n vértices. Dos vértices son adyacentes si sólo difieren en el valor de una coordenada. Puesto que hay *n* coordenadas (ejes en el cubo de *n* dimensiones), cada vértice es adyacente a otros *n* vértices. El cubo de *n* dimensiones contiene cubos de menor dimensionalidad, los cuales son vértices. Puesto que su dimensión es cero, podemos llamarlos *cubos 0*. Los bordes son cubos de dimensión 1; por tanto, los denominaremos *cubos 1*. Un lado de un cubo tridimensional es un *cubo 2*. Un cubo tridimensional entero es un *cubo 3*, etc. En general, nos referiremos a un conjunto de 2^k vértices adyacentes como un *cubo k*.

A partir de los ejemplos en las figuras 4.34 y 4.35 es claro que los *cubos k* más grandes que existen para una función son equivalentes a sus implicantes primos. A continuación veremos las técnicas de minimización que usan la representación cúbica de funciones.

4.9 UN MÉTODO TABULAR PARA MINIMIZACIÓN

La representación cúbica de las funciones lógicas es adecuada para la implementación de algoritmos de minimización que pueden programarse y ejecutarse bien en las computadoras. Las modernas herramientas CAD incluyen tales algoritmos. Si bien éstas pueden usarse eficazmente sin el conocimiento detallado de cómo se implementan sus algoritmos de minimización, tal vez parezca interesante para el lector obtener ciertos conocimientos de cómo se logra esto. En la presente sección describiremos un método tabular hasta cierto punto simple que ilustra los conceptos principales e indica algunos de los problemas que surgen.

En la década de 1950, Willard Quine [6] y Edward McCluskey [7] propusieron un enfoque tabular para la minimización. Se popularizó con el nombre de *método Quine-McCluskey*. Aunque no es lo suficientemente eficaz como para usarse en las herramientas CAD modernas, es simple e ilustra los aspectos clave. Lo presentaremos por medio de la notación cúbica expuesta en la sección 4.8.

4.9.1 GENERACIÓN DE IMPLICANTES PRIMOS

Como dijimos en la sección 4.8, los implicantes primos de una función lógica f son los cubos k más grandes posibles para los que $f = 1$. Para funciones especificadas de manera incompleta, que incluyen un conjunto de vértices no-importa, los implicantes primos son los cubos k más grandes para los que $f = 1$ o f no se especifica.

Supóngase que la especificación inicial de f está dada como mintérminos para los que $f = 1$. Además, especifiquemos los no-importa como mintérminos. Esto nos permite crear una lista de vértices para los que $f = 1$ o es una condición no-importa. Comparamos luego estos vértices por parejas para ver si pueden combinarse en cubos más grandes. Entonces podemos intentar combinar los cubos nuevos en otros todavía más grandes y continuar el proceso hasta hallar los implicantes primos.

La base del método es la propiedad combinatoria del álgebra booleana

$$x_i x_j + x_i \overline{x}_j = x_i$$

que usamos en la sección 4.8 para desarrollar la representación cúbica. Si se tienen dos cubos idénticos en todas las variables (coordenadas) excepto en una para la que un cubo tiene el valor 0 y el otro tiene 1, entonces dichos cubos pueden combinarse en un cubo más grande. Por ejemplo, considérese $f(x_1, \ldots, x_4) = \{1000, 1001, 1010, 1011\}$. Los cubos 1000 y 1001 sólo difieren en la variable x_4; pueden combinarse en un cubo nuevo 100x. De manera similar, 1010 y 1011 pueden combinarse en 101x. Luego 100x y 101x pueden combinarse en un cubo más grande, 10xx, que significa que es posible expresar la función simplemente como $f = x_1 \overline{x}_2$.

En la figura 4.36 se muestra cómo generar los implicantes primos para la función, f, de la figura 4.11. La función se define como

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$$

No hay condiciones no-importa. Puesto que los cubos más grandes sólo pueden generarse a partir de mintérminos que difieren sólo en una variable, es posible reducir el número de comparaciones por pares colocando los mintérminos en grupos tales que los cubos de cada uno de ellos tengan

Lista 1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
10	1 0 1 0	✓
12	1 1 0 0	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

Lista 2

0,4	0 x 0 0	✓
0,8	x 0 0 0	✓
8,10	1 0 x 0	
4,12	x 1 0 0	✓
8,12	1 x 0 0	✓
10,11	1 0 1 x	
12,13	1 1 0 x	
11,15	1 x 1 1	
13,15	1 1 x 1	

Lista 3

0,4,8,12	x x 0 0

Figura 4.36 Generación de implicantes primos para la función de la figura 4.11.

el mismo número de 1, y ordenando los grupos por el número de unos. Por tanto, será preciso comparar cada cubo de un grupo con todos los cubos del grupo inmediato anterior. En la figura 4.36 los mintérminos se ordenan de esta forma en la lista 1. (Nótese que también se indicaron los equivalentes decimales de los mintérminos para facilitar la explicación.) Los mintérminos, que también se llaman *cubos 0* como explicamos en la sección 4.8, pueden combinarse en los cubos 1 que se muestran en la lista 2. Para hacer las entradas fácilmente comprensibles indicamos los mintérminos que se combinan para formar cada cubo 1. A continuación se revisa si los cubos 0 se incluyen en los cubos 1 y se escribe una marca junto a cada cubo incluido. Ahora se generan cubos 2 a partir de los cubos 1 de la lista 2. El único cubo 2 que se genera es xx00, que se coloca en la lista 3. De nuevo, las marcas se colocan contra los cubos 1 que se incluyen en el cubo 2. Puesto que existe sólo un cubo 2, no puede haber cubos 3 para esta función. Los cubos sin marca en cada lista son los implicantes primos de *f*. Por tanto, el conjunto *P* de implicantes primos es

$$\begin{aligned} P &= \{10x0,\ 101x,\ 110x,\ 1x11,\ 11x1,\ xx00\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6\} \end{aligned}$$

4.9.2 DETERMINACIÓN DE UNA COBERTURA MÍNIMA

Tras generar el conjunto de todos los implicantes primos es preciso elegir un subconjunto de costo mínimo que abarque todos los mintérminos para los que $f = 1$. Como simple medida, supondremos que el costo es directamente proporcional al número de entradas a todas las compuertas, lo que significa al número de literales en los implicantes primos elegidos para implementar la función.

Para determinar una cobertura de costo mínimo se construye una *tabla de cobertura de implicantes primos* en la que haya una fila por cada implicante primo y una columna por cada mintérmino que deba cubrirse. Luego se colocan marcas para indicar los mintérminos cubiertos por cada implicante primo. En la figura 4.37*a* se muestra la tabla para los implicantes primos deducidos en la figura 4.36. Si hay una sola marca en alguna columna de la tabla de cobertura, entonces el implicante primo que cubre el mintérmino de esa columna es *esencial* y ha de incluirse en la cobertura final. Tal es el caso de p_6, que es el único implicante primo que cubre los mintérminos 0 y 4. El paso siguiente es eliminar la(s) fila(s) correspondiente(s) a los implicantes primos esenciales y la(s) columna(s) cubierta(s) por ellos. Por consiguiente, se eliminan p_6 y las columnas 0, 4, 8 y 12, lo que nos lleva a la tabla de la figura 4.37*b*.

Ahora podemos aplicar el concepto de *dominancia de fila* para reducir la tabla de cobertura. Obsérvese que p_1 sólo cubre el mintérmino 10, mientras que p_2 cubre tanto 10 como 11. Se dice que p_2 *domina* a p_1. Como el costo de p_2 es el mismo que el de p_1, es prudente elegir p_2 en vez de p_1, de modo que se eliminará p_1 de la tabla. De manera similar, p_5 domina a p_3; en consecuencia, se eliminará ésta de la tabla. Por tanto, se obtiene la tabla de la figura 4.37*c*, la cual indica que hay que elegir p_2 para cubrir el mintérmino 10 y p_5 para cubrir el mintérmino 13, que también se ocupa de cubrir los mintérminos 11 y 15. De este modo, la cobertura final es

$$\begin{aligned} C &= \{p_2, p_5, p_6\} \\ &= \{101x,\ 11x1,\ xx00\} \end{aligned}$$

Implicante primo	Mintérmino 0	4	8	10	11	12	13	15
$p_1 = 1\ 0\ x\ 0$			✓	✓				
$p_2 = 1\ 0\ 1\ x$				✓	✓			
$p_3 = 1\ 1\ 0\ x$						✓	✓	
$p_4 = 1\ x\ 1\ 1$					✓			✓
$p_5 = 1\ 1\ x\ 1$							✓	✓
$p_6 = x\ x\ 0\ 0$	✓	✓	✓			✓		

a) Tabla inicial de cobertura de implicantes primos

Implicante primo	Mintérmino 10	11	13	15
p_1	✓			
p_2	✓	✓		
p_3			✓	
p_4		✓		✓
p_5			✓	✓

b) Después de eliminar los implicantes primos esenciales

Implicante primo	Mintérmino 10	11	13	15
p_2	✓	✓		
p_4		✓		✓
p_5			✓	✓

c) Después de eliminar las filas dominantes

Figura 4.37 Selección de una cobertura para la función de la figura 4.11.

lo que significa que la implementación de costo mínimo de la función es

$$f = x_1\overline{x}_2x_3 + x_1x_2x_4 + \overline{x}_3\overline{x}_4$$

Ésta es la misma expresión que la derivada en la sección 4.2.2.

En este ejemplo aplicamos el concepto de dominancia de fila para reducir la tabla de cobertura. Las filas dominadas se eliminaron porque cubren menos mintérminos y el costo de sus

implicantes primos es el mismo que el de los implicantes primos de las filas dominantes. Sin embargo, no debe eliminarse una fila dominada si el costo de su implicante primo es menor que el del implicante primo de la fila que domina. En el problema 4.25 se presenta un ejemplo de esta situación.

En el ejemplo siguiente se ilustra el uso del método tabular con condiciones no-importa.

Ejemplo 4.14

Los mintérminos no-importa se incluyen en la lista inicial de la misma forma que los mintérminos para los que $f = 1$. Considere la función

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 5, 6, 7, 8, 9, 13) + D(1, 12, 15)$$

Se alienta al lector a derivar un mapa de Karnaugh para esta función como un auxiliar para visualizar la derivación que sigue. En la figura 4.38 se muestra la generación de implicantes primos, lo que produce el resultado

$$\begin{aligned} P &= \{00x0, 0x10, 011x, x00x, xx01, 1x0x, x1x1\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\} \end{aligned}$$

La tabla inicial de cobertura de implicantes primos se muestra en la figura 4.39*a*. Los mintérminos no-importa no se incluyen en la tabla porque no tienen que cubrirse. No hay implicantes primos esenciales. Al examinar esta tabla se advierte que la columna 8 tiene marcas en las mismas filas que la columna 9. Más aún, la columna 9 tiene una marca más en la fila p_5. Por tanto, la columna 9 domina a la 8. Esto se designa con el concepto de *dominancia de columna*. Cuando una columna domina a otra puede eliminarse la columna dominante, que en este caso es la

Lista 1

0	0 0 0 0	✓
1	0 0 0 1	✓
2	0 0 1 0	✓
8	1 0 0 0	✓
5	0 1 0 1	✓
6	0 1 1 0	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

Lista 2

0,1	0 0 0 x	✓
0,2	0 0 x 0	
0,8	x 0 0 0	✓
1,5	0 x 0 1	✓
2,6	0 x 1 0	
1,9	x 0 0 1	✓
8,9	1 0 0 x	✓
8,12	1 x 0 0	✓
5,7	0 1 x 1	✓
6,7	0 1 1 x	
5,13	x 1 0 1	✓
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
7,15	x 1 1 1	✓
13,15	1 1 x 1	✓

Lista 3

0,1,8,9	x 0 0 x
1,5,9,13	x x 0 1
8,9,12,13	1 x 0 x
5,7,13,15	x 1 x 1

Figura 4.38 Generación de implicantes primos para la función del ejemplo 4.14.

Implicante primo	Mintérmino 0	2	5	6	7	8	9	13
$p_1 = 0\ 0\ x\ 0$	✓	✓						
$p_2 = 0\ x\ 1\ 0$		✓		✓				
$p_3 = 0\ 1\ 1\ x$				✓	✓			
$p_4 = x\ 0\ 0\ x$	✓					✓	✓	
$p_5 = x\ x\ 0\ 1$			✓				✓	✓
$p_6 = 1\ x\ 0\ x$						✓	✓	✓
$p_7 = x\ 1\ x\ 1$			✓		✓			✓

a) Tabla inicial de cobertura de implicantes primos

Implicante primo	Mintérmino 0	2	5	6	7	8
$p_1 = 0\ 0\ x\ 0$	✓	✓				
$p_2 = 0\ x\ 1\ 0$		✓		✓		
$p_3 = 0\ 1\ 1\ x$				✓	✓	
$p_4 = x\ 0\ 0\ x$	✓					✓
$p_5 = x\ x\ 0\ 1$			✓			
$p_6 = 1\ x\ 0\ x$						✓
$p_7 = x\ 1\ x\ 1$			✓		✓	

b) Después de eliminar las columnas 9 y 13

Implicante primo	Mintérmino 0	2	5	6	7	8
p_1	✓	✓				
p_2		✓		✓		
p_3				✓	✓	
p_4	✓					✓
p_7			✓		✓	

c) Después de eliminar las filas p_5 y p_6

Implicante primo	Mintérmino 2	6
p_1	✓	
p_2	✓	✓
p_3		✓

d) Después de incluir p_4 y p_7 en la cobertura

Figura 4.39 Selección de una cobertura para la función del ejemplo 4.14.

columna 9. Note que esto contrasta con las filas donde se eliminan las filas dominadas (en lugar de las dominantes). La razón es que cuando se elige un implicante primo para cubrir el mintérmino que corresponde a la columna dominada, este implicante primo también cubrirá el mintérmino correspondiente a la columna dominante. En el ejemplo, elegir p_4 o p_6 cubre los mintérminos 8 y 9. De manera similar, la columna 13 domina a la 5 y, por tanto, la 13 puede borrarse.

Después de eliminar las columnas 9 y 13 se obtiene la tabla reducida mostrada en la figura 4.39*b*. En ella la fila p_4 domina a la p_6 y la p_7 a la p_5. Esto significa que p_5 y p_6 pueden eliminarse, lo que resulta en la tabla de la figura 4.39*c*. Ahora, p_4 y p_7 son esenciales para cubrir los mintérminos 8 y 5, respectivamente. Por ende, se obtiene la tabla de la figura 4.39*d*, a partir de la cual es obvio que p_2 cubre los restantes mintérminos 2 y 6. Note que la fila p_2 domina las filas p_1 y p_3.

La cobertura final es

$$\begin{aligned} C &= \{p_2, p_4, p_7\} \\ &= \{0x10,\ x00x,\ x1x1\} \end{aligned}$$

y la función se implementa como

$$f = \overline{x}_1 x_3 \overline{x}_4 + \overline{x}_2 \overline{x}_3 + x_2 x_4$$

En las figuras 4.37 y 4.39 aplicamos el concepto de dominancia de fila y de columna para reducir la tabla de cobertura, pero esto no siempre es posible, como se ilustra en el ejemplo que sigue.

Ejemplo 4.15

Considere la función

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 10, 15) + D(1, 2, 7, 8, 11, 14)$$

Los implicantes primos para esta función son

$$\begin{aligned} P &= \{00xx,\ x0x0,\ x01x,\ xx11,\ 1x1x\} \\ &= \{p_1, p_2, p_3, p_4, p_5\} \end{aligned}$$

La tabla inicial de cobertura de implicantes primos se muestra en la figura 4.40*a*. No hay implicantes primos esenciales. Además, no existen filas o columnas dominantes. Más aún, todos los implicantes primos tienen el mismo costo porque cada uno de ellos se implementa con dos literales. Por tanto, la tabla no ofrece pista alguna que sirva para seleccionar una cobertura de costo mínimo.

Un buen enfoque práctico consiste en usar el concepto de *ramificación*, que se expuso en la sección 4.2.2. Se puede escoger cualquier implicante primo, digamos p_3, y primero elegir incluirlo en la cobertura final. Luego el resto de la cobertura puede determinarse en la forma usual y calcular su costo. A continuación se intenta otra posibilidad excluyendo p_3 de la cobertura final y determinando el costo resultante. Se comparan los costos y se elige la opción menos costosa.

En la figura 4.40*b* se proporciona la tabla de cobertura que queda si p_3 se incluye en la cobertura final. La tabla no incluye los mintérminos 3 y 10 porque están cubiertos por p_3. Además,

Implicante primo	Mintérmino 0	3	10	15
$p_1 = 0\ 0\ x\ x$	✓	✓		
$p_2 = x\ 0\ x\ 0$	✓		✓	
$p_3 = x\ 0\ 1\ x$		✓	✓	
$p_4 = x\ x\ 1\ 1$		✓		✓
$p_5 = 1\ x\ 1\ x$			✓	✓

a) Tabla inicial de cobertura de implicantes primos

Implicante primo	Mintérmino 0	15
p_1	✓	
p_2	✓	
p_4		✓
p_5		✓

b) Después de incluir p_2 en la cobertura

Implicante primo	Mintérmino 0	3	10	15
p_1	✓	✓		
p_2	✓		✓	
p_4		✓		✓
p_5			✓	✓

c) Después de excluir p_2 de la cobertura

Figura 4.40 Selección de una cobertura para la función del ejemplo 4.15.

indica que una cobertura completa debe incluir p_1 o p_2 para cubrir el mintérmino 0, y p_4 o p_5 para cubrir el mintérmino 15. Por tanto, una cobertura completa puede ser

$$C = \{p_1, p_3, p_4\}$$

La alternativa de excluir p_3 conduce a la tabla de cobertura de la figura 4.40*c*. Ahí se observa que una cobertura de costo mínimo sólo requiere dos implicantes primos. Una posibilidad es elegir

p_1 y p_5; la otra es elegir p_2 y p_4. En consecuencia, una cobertura de costo mínimo es justo

$$\begin{aligned} C_{mín} &= \{p_1, p_5\} \\ &= \{00xx,\ 1x1x\} \end{aligned}$$

La función se realiza como

$$f = \overline{x}_1\overline{x}_2 + x_1x_3$$

4.9.3 RESUMEN DEL MÉTODO TABULAR

El método tabular se resume del modo siguiente:

1. A partir de una lista de cubos que representen los mintérminos donde $f = 1$ o una condición no-importa, se generan los implicantes primos mediante comparaciones por pares sucesivos de los cubos.
2. Se deriva una tabla de cobertura que indique los mintérminos donde $f = 1$ que están cubiertos por cada implicante primo.
3. Se incluyen los implicantes primos esenciales (si hay alguno) en la cobertura final y la tabla se reduce mediante la eliminación tanto de dichos implicantes primos como de los mintérminos cubiertos.
4. Se aplica el concepto de dominancia de fila y de columna para reducir aún más la tabla de cobertura. Una fila dominada se elimina sólo si el costo de su implicante primo es mayor o igual que el costo del implicante primo de la fila dominante.
5. Se repiten los pasos 3 y 4 hasta que la tabla de cobertura esté vacía o ya no sea posible reducirla más.
6. Si la tabla de cobertura reducida no está vacía, entonces se usa el enfoque de ramificación para determinar los implicantes primos restantes que han de incluirse en una cobertura de costo mínimo.

El método tabular ilustra cómo se utiliza una técnica algebraica para generar los implicantes primos. También muestra un enfoque simple para encarar el problema de la cobertura: encontrar una cobertura de costo mínimo. El método tiene ciertas limitaciones prácticas. Por citar una, las funciones rara vez se definen en la forma de mintérminos. En general se proporcionan en la forma de expresiones algebraicas o como conjuntos de cubos. La necesidad de comenzar el proceso de minimización con una lista de mintérminos implica que las expresiones o conjuntos deben expandirse de esta forma. Esta lista puede ser muy larga. Conforme se generan cubos más grandes, habrá numerosas comparaciones que hacer y los cálculos serán lentos. El uso de la tabla de cobertura para seleccionar el conjunto óptimo de implicantes primos también es computacionalmente intenso cuando se trata con funciones grandes.

Se han desarrollado muchas técnicas algebraicas cuya intención es reducir el tiempo que demoran en generarse las coberturas óptimas. Aunque el grueso de tales técnicas está más allá del ámbito de este libro, en la sección siguiente abordaremos brevemente un posible enfoque. Un lector cuya intención sea usar las herramientas CAD pero que no esté interesado en los detalles de la minimización automatizada, puede obviar esa sección sin perder la continuidad.

4.10 UNA TÉCNICA CÚBICA DE MINIMIZACIÓN

Supóngase que la especificación inicial de una función *f* está dada en términos de implicantes que no necesariamente son mintérminos o implicantes primos. Entonces es conveniente definir una operación que generará otros implicantes que no están dados de modo explícito en la especificación inicial, pero que a la postre conducirán a los implicantes primos de *f*. Una de tales posibilidades se conoce como la operación *producto* $*$, que se pronuncia como operación "producto estrella". En el texto simplemente nos referiremos a ella como *operación* $*$.

Operación $*$

La operación $*$ brinda una forma simple de derivar un cubo nuevo combinando dos cubos que sólo difieren en el valor de una variable. Sean $A = A_1A_2 \cdots A_n$ y $B = B_1B_2 \cdots B_n$ dos cubos que son implicantes de una función de n variables. Por tanto, cada coordenada A_i y B_i se especifica como poseedora del valor 0, 1 o x. La operación $*$ tiene dos pasos. Primero, la operación $*$ se evalúa para cada par A_i y B_i, en coordenadas $i = 1, 2,..., n$, de acuerdo con la tabla de la figura 4.41. Entonces, con base en los resultados obtenidos tras usar la tabla, se aplica un conjunto de reglas para determinar el resultado global de la operación $*$. En la tabla de la figura 4.41 se define la operación $*$ coordenada, $A_i * B_i$, la cual especifica el resultado de $A_i * B_i$ para cada posible combinación de valores de A_i y B_i. Este resultado es la intersección (es decir, la parte común) de A y B en esta coordenada. Nótese que cuando A_i y B_i tienen los valores opuestos (0 y 1, o viceversa), el resultado de la operación $*$ coordenada se indica mediante el símbolo ø. Se dice que la intersección de A_i y B_i está vacía. Al usar la tabla, la operación $*$ completa para A y B se define como sigue:

$$C = A * B, \text{ tal que}$$

1. $C =$ ø y $A_i * B_i =$ ø para más de una i.
2. De otro modo, $C_i = A_i * B_i$ cuando $A_i * B_i \neq$ ø, y $C_i =$ x para la coordenada donde $A_i * B_i =$ ø.

Por ejemplo, sea $A = \{0x0\}$ y $B = \{111\}$. Entonces $A_1 * B_1 = 0 * 1 =$ ø , $A_2 * B_2 =$ x $* 1 =$ 1 y $A_3 * B_3 = 0 * 1 =$ ø. Puesto que el resultado es ø en dos coordenadas, de la condición 1 se sigue que $A * B =$ ø. En otras palabras, estos dos cubos no pueden combinarse en otro, porque difieren en dos coordenadas.

Como otro ejemplo, considérese $A = \{11x\}$ y $B = \{10x\}$. En este caso, $A_1 * B_1 = 1 * 1 = 1$, $A_2 * B_2 = 1 * 0 =$ ø y $A_3 * B_3 =$ x $*$ x $=$ x. De acuerdo con la condición 2 anterior, $C_1 = 1$,

A_i \ B_i	0	1	x
0	0	ø	0
1	ø	1	1
x	0	1	x

$A_i * B_i$

Figura 4.41 Operación $*$ coordenada.

C_2 = x y C_3 = x, lo que produce $C = A * B = \{1xx\}$. A partir de dos cubos 1 que difieren sólo en una coordenada se crea un cubo 2 más grande.

El resultado de la operación $*$ puede ser un cubo más pequeño que los dos cubos implicados en la operación. Considérese $A = \{1x1\}$ y $B = \{11x\}$. Entonces $C = A * B = \{111\}$. Cabe señalar que C se incluye tanto en A como en B, lo que significa que este cubo no será útil en la búsqueda de implicantes primos. Por tanto, debe descartarse mediante el algoritmo de minimización.

Como ejemplo final, considérese $A = \{x10\}$ y $B = \{0x1\}$. Entonces $C = A * B = \{01x\}$. Los tres cubos tienen el mismo tamaño, pero C no se incluye en A ni en B. Por ende, debe considerarse en la búsqueda de implicantes primos. El lector puede encontrar útil trazar un mapa de Karnaugh para ver la manera en que el cubo C se relaciona con los cubos A y B.

Uso de la operación $*$ para hallar implicantes primos

La esencia de la operación $*$ es encontrar cubos nuevos a partir de pares de cubos existentes. En particular, es de interés encontrar los que no estén incluidos en los cubos existentes. Un procedimiento para encontrar los implicantes primos puede organizarse como sigue.

Supóngase que una función f se especifica mediante un conjunto de implicantes que están representados como cubos. Denotemos este conjunto como la cobertura C^k de f. Sean c^i y c^j cualesquiera dos cubos en C^k. Entonces apliquemos la operación $*$ a todos los pares de cubos en C^k; sea G^{k+1} el conjunto de cubos recién generados. En consecuencia

$$G^{k+1} = c^i * c^j \text{ para todo } c^i, c^j \in C^k$$

Ahora se puede formar una cobertura nueva para f empleando los cubos en C^k y G^{k+1}. Algunos de ellos pueden ser redundantes porque están incluidos en otros cubos; deben eliminarse. Sea la nueva cobertura

$$C^{k+1} = C^k \cup G^{k+1} - \text{cubos redundantes}$$

donde $\cup$ denota la unión lógica de dos conjuntos y el signo menos $(-)$ denota la eliminación de elementos de un conjunto. Si $C^{k+1} \neq C^k$, entonces se genera una nueva cobertura C^{k+2} con el mismo proceso. Si $C^{k+1} = C^k$, entonces los cubos en la cobertura son los implicantes primos de f. Para una función de n variables es preciso repetir el paso cuando mucho n veces.

Los cubos redundantes que han de eliminarse se identifican mediante la comparación por pares de cubos. El cubo $A = A_1A_2 \cdots A_n$ debe borrarse si está incluido en algún cubo $B = B_1B_2 \cdots B_n$, que es el caso si $A_i = B_i$ o B_i = x para toda coordenada i.

Ejemplo 4.16

Considere la función $f = (x_1x_2x_3)$ de la figura 4.9. Suponga que f inicialmente se especifica como un conjunto de vértices que corresponden a los mintérminos $m_0m_1m_2m_3$ y m_7. Por tanto, sea $C^0 = \{000, 001, 010, 011, 111\}$ la cobertura inicial. Con la operación $*$ para generar un conjunto nuevo de cubos se obtiene $G^1 = \{00x, 0x0, 0x1, 01x, x11\}$. Entonces $C^1 = C^0 \cup G^1$ – cubos redundantes. Observe que cada cubo en C^0 está incluido en uno de los cubos en G^1; por ende, todos los cubos en C^0 son redundantes. Así, $C^1 = G^1$.

El paso siguiente es aplicar la operación $*$ a los cubos en C^1, lo que produce $G^2 = \{000, 001, 0xx, 0x1, 010, 01x, 011\}$. Note que todos estos cubos están incluidos en el cubo 0xx; por

tanto, todos excepto 0xx son redundantes. Ahora es fácil ver que

$$C^2 = C^1 \cup G^2 - \text{términos redundantes}$$
$$= \{\text{x11, 0xx}\}$$

puesto que todos los cubos de C^1, salvo x11, son redundantes porque están cubiertos por 0xx.

Al aplicar la operación $*$ a C^2 se obtiene $G^3 = \{011\}$ y

$$C^3 = C^2 \cup G^3 - \text{términos redundantes}$$
$$= \{\text{x11, 0xx}\}$$

Como $C^3 = C^2$, la conclusión es que los implicantes primos de f son los cubos {x11, 0xx}, que representan los términos producto x_2x_3 y $\overline{x}_1$. Se trata del mismo conjunto de implicantes primos derivado por medio del mapa de Karnaugh de la figura 4.9.

Observe que la derivación de los implicantes primos en este ejemplo es similar al método tabular explicado en la sección 4.9 porque el punto de partida fue una función, f, dada como un conjunto de mintérminos.

Ejemplo 4.17 Como otro ejemplo, considere la función de cuatro variables de la figura 4.10. Suponga que esta función inicialmente se especifica como la cobertura $C^0 = \{0101, 1101, 1110, 011x, x01x\}$. Entonces las aplicaciones sucesivas de la operación $*$ y la eliminación de los términos redundantes produce

$$C^1 = \{\text{x01x, x101, 01x1, x110, 1x10, 0x1x}\}$$
$$C^2 = \{\text{x01x, x101, 01x1, 0x1x, xx10}\}$$
$$C^3 = C^2$$

Por consiguiente, los implicantes primos son $\overline{x}_2x_3$, $x_2\overline{x}_3x_4$, $\overline{x}_1x_2x_4$, $\overline{x}_1x_3$ y $x_3\overline{x}_4$.

4.10.1 DETERMINACIÓN DE LOS IMPLICANTES PRIMOS ESENCIALES

A partir de una cobertura que conste de todos los implicantes primos es necesario extraer una cobertura mínima. Como hicimos en la sección 4.2.2, todos los implicantes primos *esenciales* deben incluirse en la cobertura mínima. Para encontrarlos es útil definir una operación que determine una parte de un cubo (implicante) que *no* esté cubierta por otro cubo. Una de tales operaciones se llama *operación #* (pronúnciese "operación sharp"), que se define como sigue.

Operación #

De nuevo, sean $A = A_1 A_2 \cdots A_n$ y $B = B_1 B_2 \cdots B_n$ dos cubos (implicantes) de una función de n variables. La operación sharp $A\#B$ deja como resultado "la parte de A que no está cubierta por B". Similar a la operación $*$, la operación # tiene dos pasos: $A_i\#B_i$ se evalúa para cada coordenada i y luego se aplica un conjunto de reglas para determinar el resultado global. La opera-

A_i \ B_i	0	1	x
0	ε	ø	ε
1	ø	ε	ε
x	1	0	ε

$A_i \# B_i$

Figura 4.42 Operación # coordenada.

ción sharp para cada coordenada se define en la figura 4.42. Después de realizar esta operación para todos los pares (A_i, B_i), la operación # completa se define como sigue:

$$C = A\#B, \text{ tal que}$$

1. $C = A$ si $A_i\#B_i = \text{ø}$ para algún i.
2. $C = \text{ø}$ si $A_i\#B_i = \varepsilon$ para todo i.
3. De otro modo, $C = \bigcup_i (A_1, A_2, \ldots, \overline{B}_i, \ldots, A_n)$, donde la unión es para todo i para el que $A_i = \text{x}$ y $B_i \neq \text{x}$.

La primera condición corresponde al caso donde los cubos A y B no se intersecan, es decir, A y B difieren en el valor de al menos una variable, lo que significa que ninguna parte de A está cubierta por B. Por ejemplo, sea $A = 0\text{x}1$ y $B = 11\text{x}$. Los productos # coordenados son $A_1\#B_1 = \text{ø}$, $A_2\#B_2 = 0$ y $A_3\#B_3 = \varepsilon$. Entonces, a partir de la regla 1 se sigue que $0\text{x}1 \# 11\text{x} = 0\text{x}1$. La segunda condición refleja el caso donde A está completamente cubierta por B. Por ejemplo, $0\text{x}1 \# 0\text{xx} = \text{ø}$. La tercera condición es para el caso donde sólo una parte de A está cubierta por B. En éste, la operación # genera uno o más cubos. Específicamente, genera un cubo por cada coordenada i que sea x en A_i, pero no lo sea en B_i. Cada cubo generado es idéntico a A, excepto que A_i se sustituye por $\overline{B}_i$. Por ejemplo, $0\text{xx} \# 01\text{x} = 00\text{x}$ y $0\text{xx} \# 010 = \{00\text{x}, 0\text{x}1\}$.

Ahora mostraremos cómo usar la operación # para hallar los implicantes primos esenciales. Sea P el conjunto de todos los implicantes primos de una función dada f. Sea p^i que denota un implicante primo en el conjunto P y DC que denota los vértices no-importa para f. (En esta sección utilizaremos superíndices para referirnos a los diferentes implicantes primos porque hemos empleado subíndices para referirnos a las posiciones coordenadas en los cubos.) Entonces p^i es un implicante primo esencial si y sólo si

$$p^i \# (P - p^i) \# DC \neq \text{ø}$$

Esto significa que p^i es esencial si existe al menos un vértice para el que $f = 1$ que esté cubierto por p^i, pero no por cualquier otro implicante primo. La operación # también se realiza con el conjunto de cubos no-importa porque no es esencial cubrir los vértices en p^i que corresponden a condiciones no-importa. El significado de $p^i \# (P - p^i)$ es que la operación # se aplica sucesivamente a cada implicante primo en P. Por ejemplo, considérese $P = \{p^1, p^2, p^3, p^4\}$ y $DC = \{d^1, d^2\}$. Para comprobar que p^3 es esencial se evalúa

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

Si el resultado de esta expresión no es ø, entonces p^3 es esencial.

Ejemplo 4.18 En el ejemplo 4.16 determinamos que los cubos x11 y 0xx son los implicantes primos de la función *f* de la figura 4.9. Podemos descubrir si cada uno de dichos implicantes primos es esencial del modo siguiente

$$x11 \# 0xx = 111 \neq \emptyset$$
$$0xx \# x11 = \{00x, 0x0\} \neq \emptyset$$

El cubo x11 es esencial porque es el único implicante primo que cubre el vértice 111, para el que $f = 1$. El implicante primo 0xx es esencial porque es el único que cubre los vértices 000, 001 y 010. Esto puede observarse en el mapa de Karnaugh de la figura 4.9.

Ejemplo 4.19 En el ejemplo 4.17 encontramos que los implicantes primos de la función de la figura 4.10 son $P = \{x01x, x101, 01x1, 0x1x, xx10\}$. Puesto que esta función tiene no-importa, se calcula

$$x01x \# (P - x01x) = 1011 \neq \emptyset$$

Esto se obtuvo con los pasos siguientes: x01x # x101 = x01x, entonces x01x # 01x1 = x01x, entonces x01x # 0x1x = 101x y finalmente 101x # xx10 = 1011. De manera similar se obtiene

$$x101 \# (P - x101) = 1101 \neq \emptyset$$
$$01x1 \# (P - 01x1) = \emptyset$$
$$0x1x \# (P - 0x1x) = \emptyset$$
$$xx10 \# (P - xx10) = 1110 \neq \emptyset$$

Por tanto, los implicantes primos esenciales son x01x, x101 y xx10, pues son los únicos que cubren los vértices 1011, 1101 y 1110, respectivamente. Esto es obvio a partir del mapa de Karnaugh de la figura 4.10.

Cuando se revisa si un cubo *A* es esencial, la operación # con uno de los cubos en $P - A$ puede generar varios cubos. Si es así, entonces cada uno de ellos ha de revisarse con la operación # con todos los cubos restantes en $P - A$.

4.10.2 PROCEDIMIENTO COMPLETO PARA HALLAR UNA COBERTURA MÍNIMA

Luego de exponer las operaciones $*$ y #, ahora delinearemos un procedimiento completo para determinar la cobertura mínima para cualquier función de *n* variables. Supóngase que la función *f* se especifica en términos de vértices para los que $f = 1$; con frecuencia tales vértices se denominan el *conjunto ON* de la función. Además, supóngase que las condiciones no-importa se especifican como un *conjunto DC*. Entonces la cobertura inicial para *f* es la unión de los conjuntos ON y DC.

Los implicantes primos de *f* pueden generarse con la operación $*$, como explicamos en la sección 4.10. Luego puede aplicarse la operación # para hallar los implicantes primos esenciales como describimos en la sección 4.10.1. Si tales implicantes cubren todo el conjunto ON, entonces forman la cobertura de costo mínimo para *f*. De otro modo, es necesario incluir otros implicantes primos hasta que todos los vértices en el conjunto ON queden cubiertos.

Un implicante primo no esencial p^i debe borrarse si existe un implicante primo menos costoso p^j que cubra todos los vértices del conjunto ON cubiertos por p^i. Si los restantes implicantes primos no esenciales tienen el mismo costo, entonces un posible enfoque heurístico consiste en seleccionar arbitrariamente uno de ellos, incluirlo en la cobertura y determinar el resto de ella. Luego se genera otra cobertura excluyendo este implicante primo y se elige la implementación de la cobertura de costo más bajo. Ya utilizamos este enfoque, que a menudo se conoce como *ramificación heurística*, en las secciones 4.2.2 y 4.9.2.

La explicación anterior puede resumirse en la forma del procedimiento de minimización presentado enseguida:

1. Sea $C^0 = ON \cup DC$ la cobertura inicial de la función f y sus condiciones no-importa.
2. Se encuentran todos los implicantes primos de C^0 con la operación $*$; sea P este conjunto de implicantes primos.
3. Se determinan los implicantes primos esenciales mediante la operación #. Un implicante primo p^i es esencial si $p^i \# (P - p^i) \# DC \neq \emptyset$. Si los implicantes primos esenciales cubren todos los vértices del conjunto ON, entonces forman la cobertura de costo mínimo.
4. Se borra cualquier p^i no esencial que sea más costoso (es decir, un cubo más pequeño) que algún otro implicante primo p^j si $p^i \# DC \# p^j = \emptyset$.
5. Se eligen los implicantes primos de menor costo para cubrir los vértices restantes del conjunto ON. Se aplica la ramificación heurística en los implicantes primos de igual costo y se conserva la cobertura con el costo más bajo.

Ejemplo 4.20

Para ilustrar el procedimiento de minimización emplearemos la función

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

Para ayudar al lector a seguir la explicación, esta función también se muestra en la forma de un mapa de Karnaugh en la figura 4.43.

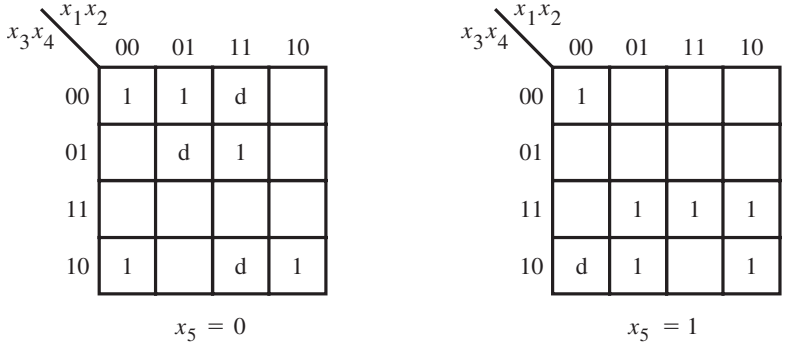


Figura 4.43 La función del ejemplo 4.20.

En lugar de que *f* se especifique en términos de mintérminos, suponga que *f* está dada como la siguiente expresión en SOP

$$f = \overline{x}_1\overline{x}_3\overline{x}_4\overline{x}_5 + x_1x_2\overline{x}_3x_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4x_5 + \overline{x}_1x_2x_3x_5 + x_1\overline{x}_2x_3x_5 + x_1x_3x_4x_5 + \overline{x}_2x_3\overline{x}_4\overline{x}_5$$

Además, supondremos que los no-importa se especifican con la expresión

$$\text{DC} = x_1x_2\overline{x}_4\overline{x}_5 + \overline{x}_1x_2\overline{x}_3x_4\overline{x}_5 + \overline{x}_1\overline{x}_2x_3\overline{x}_4x_5$$

Por tanto, el conjunto ON expresado como cubo es

$$\text{ON} = \{0\text{x}000,\ 11010,\ 00001,\ 011\text{x}1,\ 101\text{x}1,\ 1\text{x}111,\ \text{x}0100\}$$

y el conjunto de no-importa es

$$\text{DC} = \{11\text{x}00,\ 01010,\ 00101\}$$

La cobertura inicial C^0 consta de los conjuntos ON y DC:

$$C^0 = \{0\text{x}000,\ 11010,\ 00001,\ 011\text{x}1,\ 101\text{x}1,\ 1\text{x}111,\ \text{x}0100,\ 11\text{x}00,\ 01010,\ 00101\}$$

Al usar la operación $*$, las subsecuentes coberturas que se obtienen son

$$C^1 = \{0\text{x}000,\ 011\text{x}1,\ 101\text{x}1,\ 1\text{x}111,\ \text{x}0100,\ 11\text{x}00,\ 0000\text{x},\ 00\text{x}00,\ \text{x}1000,\ 010\text{x}0,\ 110\text{x}0,\ \text{x}1010,\ 00\text{x}01,\ \text{x}1111,\ 0\text{x}101,\ 1010\text{x},\ \text{x}0101,\ 1\text{x}100,\ 0010\text{x}\}$$

$$C^2 = \{0\text{x}000,\ 011\text{x}1,\ 101\text{x}1,\ 1\text{x}111,\ 11\text{x}00,\ \text{x}1111,\ 0\text{x}101,\ 1\text{x}100,\ \text{x}010\text{x},\ 00\text{x}0\text{x},\ \text{x}10\text{x}0\}$$

$$C^3 = C^2$$

En consecuencia, $P = C^2$.

Al usar la operación # se encuentra que hay dos implicantes primos esenciales: 00x0x (porque es el único que cubre el vértice 00001) y x10x0 (porque es el único que abarca el vértice 11010). Los mintérminos de *f* cubiertos por estos dos implicantes primos son $m(0, 1, 4, 8, 26)$.

A continuación, se encuentra que 1x100 puede eliminarse porque el único vértice del conjunto ON que cubre es 10100 (m_{20}), que también está cubierto por x010x y el costo de este implicante primo es más bajo. Observe que tras remover 1x100, el implicante primo x010x se convierte en esencial porque ninguno de los otros cubre el vértice 10100. En consecuencia, x010x ha de incluirse en la cobertura final. Cubre $m(20, 21)$.

Aún falta encontrar implicantes primos para cubrir $m(13, 15, 23, 31)$. Mediante el empleo de ramificación heurística, la cobertura de costo más bajo se obtiene incluyendo los implicantes primos 011x1 y 1x111. Por tanto, la cobertura final es

$$C_{mínima} = \{00\text{x}0\text{x},\ \text{x}10\text{x}0,\ \text{x}010\text{x},\ 011\text{x}1,\ 1\text{x}111\}$$

La correspondiente expresión en suma de productos es

$$f = \overline{x}_1\overline{x}_2\overline{x}_4 + x_2\overline{x}_3\overline{x}_5 + \overline{x}_2x_3\overline{x}_4 + \overline{x}_1x_2x_3x_5 + x_1x_3x_4x_5$$

Aunque este procedimiento resulta tedioso cuando se realiza a mano, no es difícil escribir un programa de computadora para implementar automáticamente el algoritmo. El lector debe comprobar la validez de la solución hallando la realización óptima del mapa de Karnaugh de la figura 4.43.

4.11 CONSIDERACIONES PRÁCTICAS

El propósito de la sección anterior fue brindar al lector cierta idea de cómo automatizar la minimización de las funciones lógicas para utilizarla en las herramientas CAD. Elegimos un esquema no muy difícil de explicar. Desde el punto de vista práctico, este esquema tiene ciertas desventajas. La principal es que el número de cubos que deben considerarse en el proceso puede ser muy grande.

Si la meta de la minimización se relaja de manera que no sea imperativo encontrar una implementación de costo mínimo, entonces es posible derivar técnicas heurísticas que produzcan buenos resultados en tiempo razonable. Una técnica de este tipo forma la base del ampliamente usado programa Espresso, que está disponible por Internet en la Universidad de California en Berkeley. Espresso es un programa de optimación en dos niveles. Tanto su entrada como su salida se especifican en el formato de cubos. En vez de usar la operación $*$ para hallar los implicantes primos, Espresso aplica una técnica de expansión de implicantes (véase, en el problema 4.30, una ilustración de la expansión de implicantes). Una explicación completa de Espresso se brinda en [19], mientras que bosquejos simplificados se encuentran en [3, 12].

La Universidad de California en Berkeley también ofrece dos programas, llamados MIS [20] y SIS [21], que sirven para diseñar circuitos multinivel. Ambos permiten que el usuario aplique varias técnicas de optimación multinivel a un circuito lógico. El usuario puede experimentar con diferentes estrategias de optimación mediante la aplicación de técnicas como la factorización y la descomposición a todo o parte de un circuito. SIS también incluye el algoritmo de Espresso para minimizar funciones en dos niveles, así como muchas otras técnicas de optimación.

En el mercado se encuentran varios sistemas CAD. Cuatro compañías cuyos productos se usan mucho son Cadence Design Systems, Mentor Graphics, Synopsys y Synplicity. La información de sus productos está disponible en la web. Cada compañía ofrece software de síntesis lógico que sirve para centrarse en varios tipos de chips, como PLD, arreglos de compuertas, celdas estándar y chips a la medida. Puesto que hay muchas posibles formas de sintetizar un circuito, como vimos en las secciones previas, cada producto comercial utiliza una estrategia de optimación lógica patentada con base en la heurística.

Para describir las herramientas CAD se han inventado términos nuevos. En particular cabe mencionar dos que se usan mucho en la industria: *síntesis lógica independiente de tecnología* y *mapeo de tecnología*. El primero se refiere a las técnicas aplicadas cuando se optima un circuito sin considerar los recursos disponibles en el chip destino. La mayor parte de las técnicas presentadas en este capítulo es de este tipo. El segundo término, mapeo de tecnología, se refiere a las técnicas empleadas para garantizar que el circuito producido por la síntesis lógica puede realizarse con los recursos lógicos disponibles en el chip destino. Un buen ejemplo de mapeo de tecnología es la transformación de un circuito en la forma de operaciones lógicas como AND y OR en otro que conste sólo de operaciones NAND. Este tipo de mapeo de tecnología se lleva a cabo cuando se dirige un circuito a un arreglo de compuertas que contiene sólo compuertas NAND. Otro ejemplo es la traducción de operaciones lógicas en tablas de consulta, el cual se realiza cuando se dirige un diseño a un FPGA.

En el capítulo 12 se explican las herramientas CAD pormenorizadamente. Se presenta un flujo de diseño típico que puede usar un diseñador para implementar un sistema digital.

4.12 EJEMPLOS DE CIRCUITOS SINTETIZADOS A PARTIR DE CÓDIGO DE VHDL

En la sección 2.10 mostramos cómo escribir programas simples en VHDL para describir funciones lógicas. En esta sección se presentan características adicionales de VHDL y se ofrecen más ejemplos de circuitos diseñados con código de ese lenguaje.

Recuérdese que una señal lógica se representa en VHDL como un objeto de datos, y cada objeto de datos tiene un tipo asociado. En los ejemplos de la sección 2.10 todos los objetos de datos tienen el tipo BIT, lo que significa que sólo pueden asumir los valores 0 y 1. Para dar más flexibilidad, VHDL proporciona otro tipo de datos llamado *STD_LOGIC*. Las señales representadas con este tipo pueden tener varios valores.

Como su nombre lo indica, STD_LOGIC busca servir como el tipo de datos estándar para la representación de señales lógicas. En la figura 4.44 se presenta un ejemplo que usa el tipo STD_LOGIC. La expresión lógica para *f* corresponde a la tabla de verdad de la figura 4.1; dicha tabla describe *f* en la forma canónica, que consiste en mintérminos. Para usar el tipo STD_LOGIC, el código de VHDL debe incluir las dos líneas dadas al principio de la figura. Tales instrucciones sirven como directivas al compilador de VHDL. Son necesarias porque el estándar original de VHDL, IEEE 1076, no incluye el tipo STD_LOGIC. La forma en que el nuevo tipo se agregó al lenguaje, en el estándar IEEE 1164, consistió en proporcionar la definición de STD_LOGIC como un conjunto de archivos que pueden incluirse con código de VHDL cuando se compile. El conjunto de archivos se llama *biblioteca* (*library*). El propósito de la primera línea de la figura 4.44 es declarar que el código usará la biblioteca IEEE.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func1 IS
    PORT ( x1, x2, x3  : IN   STD_LOGIC ;
           f           : OUT  STD_LOGIC ) ;
END func1 ;

ARCHITECTURE LogicFunc OF func1 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND NOT x3) OR
         (NOT x1 AND x2 AND NOT x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figura 4.44 Código de VHDL para la función de la figura 4.1.

En VHDL hay dos aspectos principales para la definición de un tipo de datos nuevo. Primero debe especificarse el conjunto de valores que un objeto de datos del tipo nuevo puede asumir. Para STD_LOGIC existen varios valores legales, pero los más importantes para describir las funciones lógicas son 0, 1, Z y –. El valor Z, que representa el estado de alta impedancia, se explicó en la sección 3.8.8. El valor lógico – representa la condición no-importa, que marcamos con *d* en la sección 4.4. El segundo requisito es que deben especificarse todos los usos legales del nuevo tipo de datos en código de VHDL. Por ejemplo, es necesario especificar que el tipo STD_LOGIC es legal para usarlo con operadores booleanos.

En la biblioteca IEEE uno de los archivos define el tipo de datos STD_LOGIC mismo y especifica algunos usos legales básicos, como para las operaciones booleanas. En la figura 4.44 la segunda línea del código indica al compilador de VHDL que use las definiciones en este archivo cuando compile el código. El archivo encapsula la definición de STD_LOGIC en lo que se conoce como *paquete*. El paquete se llama std_logic_1164. Es posible instruir al compilador de VHDL para que utilice sólo un subconjunto del paquete, pero el uso normal es especificar la palabra *all* para indicar que todo el paquete es de interés, como se hizo en la figura.

Para los ejemplos de código de VHDL presentados en este libro, casi siempre se usará únicamente el tipo STD_LOGIC. Además de simplificar el código, ocupar sólo un tipo de datos tiene otro beneficio. VHDL es un lenguaje con una enorme comprobación de tipos, lo que significa que el compilador verifica cuidadosamente todas las instrucciones de asignación de objetos de datos para asegurar que el tipo del objeto del lado izquierdo de la instrucción de asignación sea exactamente el mismo que el tipo del objeto de datos del lado derecho. Incluso si dos objetos de datos parecen compatibles desde el punto de vista intuitivo, digamos un objeto de tipo BIT y otro de tipo STD_LOGIC, el compilador no permitirá que uno se asigne al otro. Muchas herramientas de síntesis proporcionan pequeños programas de conversión para convertir —valga la redundancia— de un tipo a otro, pero este conflicto se evita usando sólo el tipo de datos STD_LOGIC en la mayoría de los casos. En el resto de la sección presentamos algunos ejemplos de código de VHDL. Mostraremos los resultados de sintetizar el código para implementarlo en dos tipos de chips, un CPLD y un FPGA.

Ejemplo 4.21

Se compiló el código de VHDL de la figura 4.44 para implementarlo en un CPLD, y las herramientas CAD produjeron la expresión

$$f = \overline{x}_3 + x_1\overline{x}_2$$

que es la forma mínima en suma de productos que se derivó con el mapa de Karnaugh de la figura 4.5*b*. En la figura 4.45 se muestra cómo implementar esta expresión en un CPLD. Los interruptores programados para estar cerrados se muestran en gris, lo mismo que las compuertas empleadas para implementar *f*. Observe que, en este caso, sólo se usan las dos compuertas AND superiores. Las tres compuertas AND de la parte inferior no tienen efecto porque cada una está conectada tanto a la versión verdadera como a la complementada de una entrada no utilizada, lo que ocasiona que la salida de la compuerta AND sea 0.

En la figura 4.46 se dan los resultados de sintetizar el código de VHDL de la figura 4.44 en un FPGA. Se supone que el compilador genera la misma forma en suma de productos que la expuesta líneas arriba. Puesto que las celdas lógicas en el chip son tablas de consulta de cuatro entradas, sólo se necesita una celda lógica para esta función. En la figura se muestra que las va-

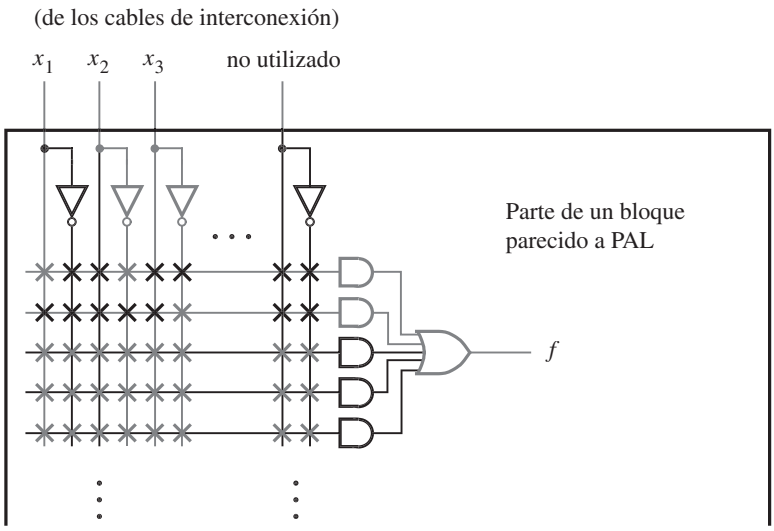
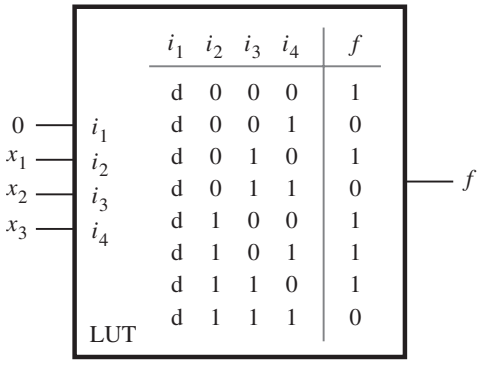


Figura 4.45 Implementación del código de VHDL de la figura 4.44.



i_1	i_2	i_3	i_4	f
d	0	0	0	1
d	0	0	1	0
d	0	1	0	1
d	0	1	1	0
d	1	0	0	1
d	1	0	1	1
d	1	1	0	1
d	1	1	1	0

Figura 4.46 El código de VHDL de la figura 4.44 implementado en una LUT.

riables x_1, x_2 y x_3 están conectadas a las entradas de la LUT llamadas i_2, i_3 e i_4. La entrada i_1 no se usa porque la función sólo requiere tres entradas. La tabla de verdad de la LUT indica que la entrada no usada se trata como un no-importa. Por tanto, sólo se muestra la mitad de las filas de la tabla, pues la otra mitad es idéntica. La entrada no usada en la LUT se muestra conectada a 0 en la figura, pero también podría conectarse a 1.

Es interesante considerar los beneficios ofrecidos por las optimaciones aplicadas en la síntesis lógica. Para la implementación en el CPLD, la función se simplificó de los cinco términos

producto originales en la forma canónica a sólo dos términos producto. Sin embargo, tanto la forma optimizada como la no optimizada encajan en una macrocelda individual en el chip y, por tanto, tienen el mismo costo (la macrocelda de la figura 4.45 tiene cinco términos producto). De manera similar, para el FPGA, no importa si la función se minimiza, ya que encaja en una sola LUT. La razón es que el circuito de nuestro ejemplo es muy pequeño. Para circuitos grandes es esencial llevar a cabo la optimación. En los ejemplos 4.22 y 4.23 se ilustran funciones lógicas para las que el costo de implementación se reduce cuando se optima.

Ejemplo 4.22

El código de VHDL de la figura 4.47 corresponde a la función f_1 de la figura 4.7. Como existen seis términos producto en la forma canónica, se necesitarían dos macroceldas del tipo de la figura 4.45. Cuando se sintetice mediante herramientas CAD, la expresión resultante podría ser

$$f = \overline{x}_2 x_3 + x_1 \overline{x}_3 x_4$$

que es la misma que la expresión derivada en la figura 4.7. Puesto que la expresión optimizada sólo tiene dos términos producto, puede realizarse con sólo una macrocelda y, por ende, resulta en un costo más bajo.

Cuando f_1 se sintetiza para implementarla en un FPGA, la expresión generada puede ser la misma que para el CPLD. Como la función sólo tiene cuatro entradas, únicamente necesita una LUT.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func2 IS
    PORT ( x1, x2, x3, x4  : IN     STD_LOGIC ;
           f               : OUT  STD_LOGIC ) ;
END func2 ;

ARCHITECTURE LogicFunc OF func2 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND x3 AND NOT x4) OR
         (NOT x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND NOT x2 AND NOT x3 AND x4) OR
         (x1 AND NOT x2 AND x3 AND NOT x4) OR
         (x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND x2 AND NOT x3 AND x4) ;
END LogicFunc ;
```

Figura 4.47 Código de VHDL para f_1 de la figura 4.7.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func3 IS
    PORT ( x1, x2, x3, x4, x5, x6, x7  : IN     STD_LOGIC ;
           f                           : OUT  STD_LOGIC ) ;
END func3 ;

ARCHITECTURE LogicFunc OF func3 IS
BEGIN
    f <= (x1 AND x3 AND NOT x6) OR
         (x1 AND x4 AND x5 AND NOT x6) OR
         (x2 AND x3 AND x7) OR
         (x2 AND x4 AND x5 AND x7) ;
END LogicFunc ;
```

Figura 4.48 Código de VHDL para la función de la sección 4.7.

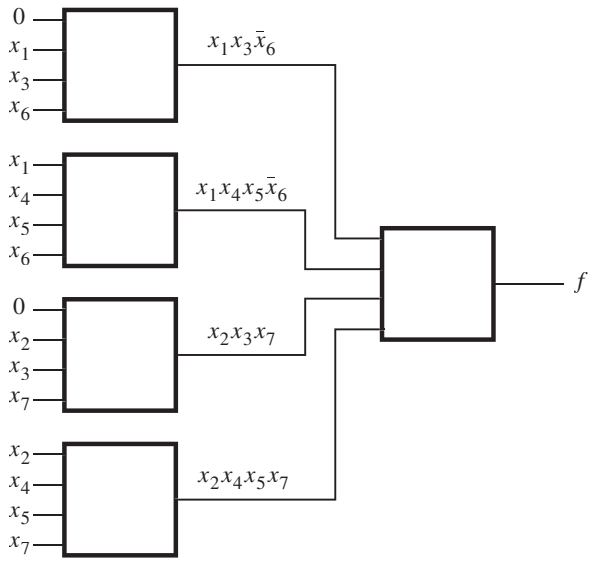
Ejemplo 4.23 En la sección 4.6 usamos una función lógica de siete variables como una motivación para la síntesis en multinivel. Esta función está dada en el código de VHDL de la figura 4.48. La expresión lógica se halla en forma mínima de suma de productos. Cuando se sintetiza para implementarla en un CPLD, las herramientas CAD no realizan optimaciones. La función requiere una macrocelda. Esta función es más interesante cuando se considera su implementación en un FPGA con LUT de cuatro entradas. Puesto que existen siete entradas, se requiere más de una LUT. Si la función se implementa directamente como se da en el código de VHDL, entonces se necesitan cinco LUT, como se muestra en la figura 4.49*a*. En lugar de mostrar la tabla de verdad programada en cada LUT, se presenta la función lógica que se implementó en la salida de la LUT. Pero si la función se sintetiza como

$$f = (x_1\overline{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

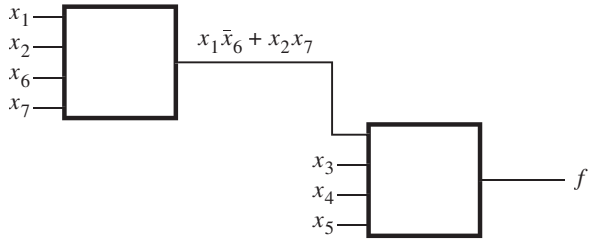
que es la expresión que derivamos mediante factorización en la sección 4.6, entonces f puede implementarse sólo con dos LUT, como se ilustra en la figura 4.49*b*. Una LUT produce el término $S = x_1\overline{x}_6 + x_2x_7$; la otra implementa la función de cuatro entradas $f = Sx_3 + Sx_4x_5$.

4.13 COMENTARIOS FINALES

La intención de este capítulo fue ofrecer al lector algunas ideas de diversos aspectos de la síntesis de funciones lógicas. Ahora que el lector se siente cómodo con los conceptos fundamentales, podemos examinar circuitos digitales de naturaleza más refinada. En el capítulo siguiente describimos circuitos que realizan operaciones aritméticas, que son una parte clave de las computadoras.



a) Realización en suma de productos



b) Realización factorizada

Figura 4.49 Implementación del código de VHDL de la figura 4.48.

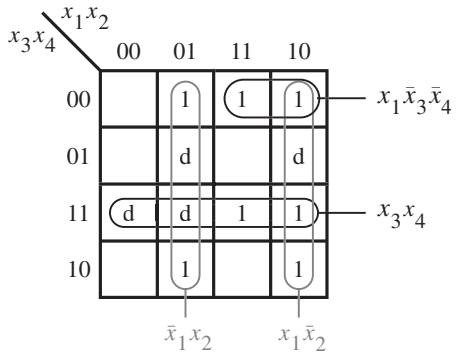
4.14 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección presentamos algunos problemas típicos que el lector puede encontrar y mostramos cómo resolverlos.

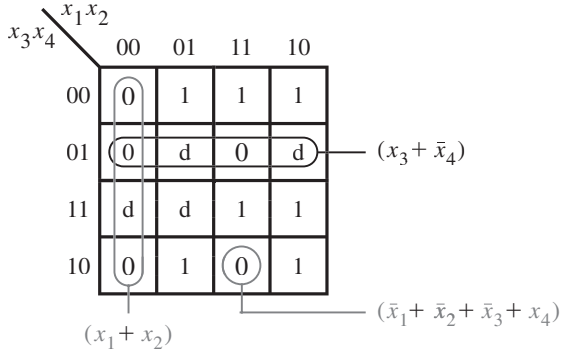
Ejemplo 4.24

Problema: Determine las expresiones en SOP y POS de costo mínimo para la función $f(x_1, x_2, x_3, x_4) = \sum m(4, 6, 8, 10, 11, 12, 15) + D(3, 5, 7, 9)$.

Solución: La función puede representarse en la forma de mapa de Karnaugh como se muestra en la figura 4.50*a*. Observe que la ubicación de los mintérminos en el mapa es como se indica en la figura 4.6.



a) Determinación de la expresión en SOP



b) Determinación de la expresión en POS

Figura 4.50 Mapas de Karnaugh para el ejemplo 4.24.

Para encontrar la expresión en SOP de costo mínimo es preciso hallar los implicantes primos que cubran todos los 1 del mapa. Los no-importa pueden usarse según se desee. El mintérmino m_6 sólo queda cubierto por el implicante primo $\overline{x}_1x_2$; por tanto, este implicante es esencial y debe incluirse en la expresión final. De manera similar, los implicantes primos $x_1\overline{x}_2$ y x_3x_4 son esenciales porque son los únicos que cubren m_{10} y m_{15}, respectivamente. Estos tres implicantes primos cubren todos los mintérminos para los que $f = 1$, excepto m_{12}. Este mintérmino puede cubrir de dos formas, al elegir $x_1\overline{x}_3\overline{x}_4$ o $x_2\overline{x}_3\overline{x}_4$. Como ambos implicantes primos tienen el mismo costo, puede elegirse cualquiera de ellos. Si es el primero, la expresión en SOP deseada es

$$f = \overline{x}_1x_2 + x_1\overline{x}_2 + x_3x_4 + x_1\overline{x}_3\overline{x}_4$$

Estos implicantes primos se encierran en círculos en el mapa.

La expresión en POS deseada puede encontrarse como se indica en la figura 4.50*b*. En este caso, hay que hallar los términos suma que cubren todos los 0 en la función. Nótese que escribimos 0 explícitamente en el mapa para destacar este hecho. El término $(x_1 + x_2)$ es esencial para cubrir los 0 en los cuadrados 0 y 2, que corresponden a los maxitérminos M_0 y M_2. Los términos $(x_3 + \overline{x}_4)$ y $(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$ deben usarse para cubrir los 0 en los cuadrados 13 y 14, respectivamente. Como estos tres términos suma cubren todos los 0 del mapa, la expresión en POS es

$$f = (x_1 + x_2)(x_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$$

Los términos suma elegidos se encierran en círculos en el mapa.

Observe el uso de los no-importa en este ejemplo. Para obtener una expresión en SOP de costo mínimo hemos supuesto que los cuatro no-importa tienen el valor 1. Pero la expresión en POS de costo mínimo sólo se vuelve posible si suponemos que los no-importa 3, 5 y 9 tienen el valor 0, en tanto que el no-importa 7 tiene el valor 1. Esto significa que las expresiones en SOP y POS resultantes no son idénticas en términos de las funciones que representan. Cubren de manera idéntica todas las combinaciones para las que *f* se especifica como 1 o 0, pero difieren en las combinaciones 3, 5 y 9. Desde luego, esta diferencia no tiene importancia, porque las combinaciones no-importa nunca se aplicarán como entradas a los circuitos implementados.

Ejemplo 4.25

Problema: Use mapas de Karnaugh para encontrar las expresiones en SOP y POS de costo mínimo para la función

$$f(x_1, \ldots, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$$

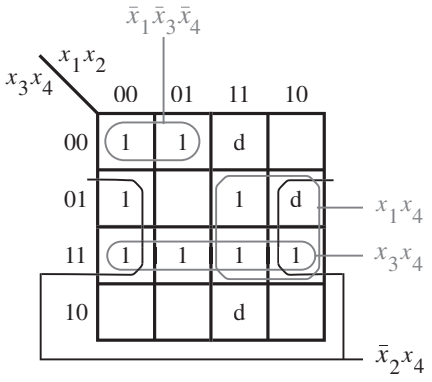
si se supone que también existen no-importa definidos como $D = \sum(9, 12, 14)$.

Solución: El mapa de Karnaugh que representa esta función se muestra en la figura 4.51*a*. El mapa se deriva colocando 1 que correspondan a cada término producto en la expresión usada para especificar *f*. El término $\overline{x}_1\overline{x}_3\overline{x}_4$ corresponde a los mintérminos 0 y 4. El término x_3x_4 representa la tercera fila del mapa, que comprende los mintérminos 3, 7, 11 y 15. El término $\overline{x}_1\overline{x}_2x_4$ especifica los mintérminos 1 y 3. El cuarto término producto representa el mintérmino 13. El mapa también incluye las tres condiciones no-importa.

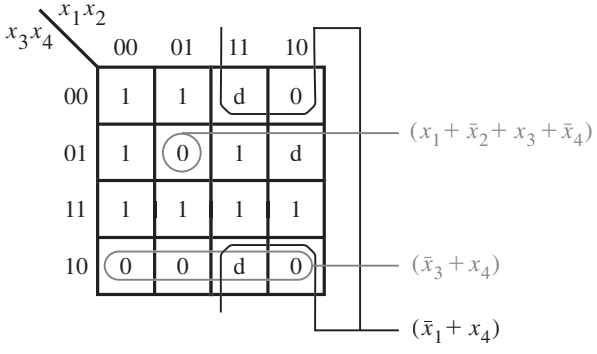
Para hallar la expresión en SOP deseada hay que encontrar el conjunto de implicantes primos menos costoso que cubra todos los 1 en el mapa. El término x_3x_4 es un implicante primo que debe incluirse porque es el único que cubre el mintérmino 7; también cubre los mintérminos 3, 11 y 15. El mintérmino 4 puede cubrirse con $\overline{x}_1\overline{x}_3\overline{x}_4$ o con $x_2\overline{x}_3\overline{x}_4$. Ambos términos tienen el mismo costo; se elegirá $\overline{x}_1\overline{x}_3\overline{x}_4$ porque también cubre el mintérmino 0. El mintérmino 1 puede cubrirse con $\overline{x}_1\overline{x}_2\overline{x}_3$ o con $\overline{x}_2x_4$; se debe elegir el último porque su costo es menor. Esto sólo deja por cubrir el mintérmino 13, lo que puede hacerse con x_1x_4 o con x_1x_2 a costos iguales. Al elegir x_1x_4 la expresión en SOP de costo mínimo es

$$f = x_3x_4 + \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2x_4 + x_1x_4$$

En la figura 4.51*b* se muestra cómo encontrar la expresión en POS. El término suma $(\overline{x}_3 + x_4)$ cubre los 0 en la fila inferior. Para cubrir el 0 en el cuadrado 8 hay que incluir $(\overline{x}_1 + x_4)$. Los 0



a) Determinación de la expresión en SOP



b) Determinación de la expresión en POS

Figura 4.51 Mapas de Karnaugh para el ejemplo 4.25.

restantes, en el cuadrado 5, deben cubrirse con $(x_1 + \overline{x}_2 + x_3 + \overline{x}_4)$. Por tanto, la expresión en POS de costo mínimo es

$$f = (\overline{x}_3 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + x_3 + \overline{x}_4)$$

Ejemplo 4.26 **Problema:** Utilice el método tabular expuesto en la sección 4.9 para derivar una expresión en SOP de costo mínimo para la función

$$f(x_1, \ldots, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$$

si se supone que también existen no-importa definidos como $D = \sum(9, 12, 14)$.

Solución: Con el método tabular es necesario comenzar con la función definida en la forma de mintérminos. Como se encontró en la figura 4.51*a*, la función *f* también puede representarse como

$$f(x_1, \ldots, x_4) = \sum m(0, 1, 3, 4, 7, 11, 13, 15) + D(9, 12, 14)$$

Los correspondientes 11 cubos 0 se colocan en la lista 1 de la figura 4.52.

Ahora, realice una comparación por pares de todos los cubos 0 para determinar los cubos 1 que se muestran en la lista 2, los cuales se obtienen al combinar pares de cubos 0. Note que todos los cubos 0 se incluyen en los cubos 1, como se indica con las marcas en la lista 1. A continuación, realice una comparación por pares de todos los cubos 1 para obtener los cubos 2 en la lista 3. Algunos de dichos cubos 2 pueden generarse de varias formas, pero no es útil listar mas de una vez cubos 2 (por ejemplo, x0x1 en la lista 3 puede obtenerse combinando de la lista 2 los cubos 1, 3 y 9, 11, o si se usan los cubos 1, 9 y 3, 11). Note que todos los cubos 1, excepto tres, se incluyen en los cubos 2. No es posible generar algún cubo 3; por tanto, todos los términos que no se incluyen en algún otro término (los términos sin marca en la lista 2 y todos los términos en la lista 3) son los implicantes primos de *f*. El conjunto de implicantes primos es

$$\begin{aligned} P &= \{000x, 0x00, x100, x0x1, xx11, 1xx1, 11xx\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\} \end{aligned}$$

Para encontrar la cobertura de costo mínimo para *f*, construya la tabla de la figura 4.53*a* que muestra todos los implicantes primos y los mintérminos que deben cubrirse: aquellos para los que $f = 1$. Para indicar que un mintérmino se cubre mediante un implicante primo en particular se coloca una marca. Puesto que el mintérmino 7 se cubre sólo por p_5, este implicante primo

Lista 1

0	0 0 0 0	✓
1	0 0 0 1	✓
4	0 1 0 0	✓
3	0 0 1 1	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
14	1 1 1 0	✓
15	1 1 1 1	✓

Lista 2

0,1	0 0 0 x	
0,4	0 x 0 0	
1,3	0 0 x 1	✓
1,9	x 0 0 1	✓
4,12	x 1 0 0	
3,7	0 x 1 1	✓
3,11	x 0 1 1	✓
9,11	1 0 x 1	✓
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
12,14	1 1 x 0	✓
7,15	x 1 1 1	✓
11,15	1 x 1 1	✓
13,15	1 1 x 1	✓
14,15	1 1 1 x	✓

Lista 3

1,3,9,11	x 0 x 1
3,7,11,15	x x 1 1
9,11,13,15	1 x x 1
12,13,14,15	1 1 x x

Figura 4.52 Generación de implicantes primos para la función del ejemplo 4.26.

Implicante primo	Mintérmino 0	1	3	4	7	11	13	15
p_1 = 0 0 0 x	✓	✓						
p_2 = 0 x 0 0	✓			✓				
p_3 = x 1 0 0				✓				
p_4 = x 0 x 1		✓	✓			✓		
p_5 = x x 1 1			✓		✓	✓		✓
p_6 = 1 x x 1						✓	✓	✓
p_7 = 1 1 x x							✓	✓

a) Tabla inicial de cobertura de implicantes primos

Implicante primo	Mintérmino 0	1	4	13
p_1 = 0 0 0 x	✓	✓		
p_2 = 0 x 0 0	✓		✓	
p_4 = x 0 x 1		✓		
p_6 = 1 x x 1				✓

b) Después de eliminar las filas p_3, p_5, y p_7, y las columnas 3, 7, 11 y 13

Figura 4.53 Selección de una cobertura para la función del ejemplo 4.26.

debe incluirse en la cobertura final. Observe que la fila p_2 domina a la p_3, por lo que esta última puede eliminarse. De manera similar, la fila p_6 domina a la p_7. La eliminación de las filas p_5, p_3 y p_7, así como de las columnas 3, 7, 11 y 15 (que están cubiertas por p_5) conduce a la tabla reducida de la figura 4.53*b*. En ella p_2 y p_6 son esenciales. Cubren los mintérminos 0, 4 y 13. Por consiguiente, sólo falta por cubrir el mintérmino 1, lo que se hace eligiendo p_1 o p_4. Como p_4 tiene un costo más bajo, debe elegirse. En consecuencia, la cobertura final es

$$\begin{aligned} C &= \{p_2, p_4, p_5, p_6\} \\ &= \{0x00,\ x0x1,\ xx11,\ 1xx1\} \end{aligned}$$

y la función se implementa como

$$f = \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2x_4 + x_3x_4 + x_1x_4$$

Ejemplo 4.27

Problema: Use la operación producto $*$ para encontrar todos los implicantes primos de la función

$$f(x_1, \ldots, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$$

si se supone que también existen no-importa definidos como $D = \sum(9, 12, 14)$.

Solución: El conjunto ON para esta función es

$$\text{ON} = \{0\text{x}00, \text{xx}11, 00\text{x}1, 1101\}$$

La cobertura inicial, que consta del conjunto ON y los no-importa, es

$$C^0 = \{0\text{x}00, \text{xx}11, 00\text{x}1, 1101, 1001, 1100, 1110\}$$

Al usar la operación $*$, las subsecuentes coberturas obtenidas son

$$C^1 = \{0\text{x}00, \text{xx}11, 00\text{x}1, 000\text{x}, \text{x}100, 11\text{x}1, 10\text{x}1, 111\text{x}, \text{x}001, 1\text{x}01, 110\text{x}, 11\text{x}0\}$$
$$C^2 = \{0\text{x}00, \text{xx}11, 000\text{x}, \text{x}100, \text{x}0\text{x}1, 1\text{xx}1, 11\text{xx}\}$$
$$C^3 = C^2$$

Por tanto, el conjunto de todos los implicantes primos es

$$P = \{\overline{x}_1\overline{x}_3\overline{x}_4, x_3x_4, \overline{x}_1\overline{x}_2\overline{x}_3, x_2\overline{x}_3\overline{x}_4, \overline{x}_2x_4, x_1x_4, x_1x_2\}$$

Ejemplo 4.28

Problema: Encuentre la implementación de costo mínimo para la función

$$f(x_1, \ldots, x_4) = \overline{x}_1\overline{x}_3\overline{x}_4 + x_3x_4 + \overline{x}_1\overline{x}_2x_4 + x_1x_2\overline{x}_3x_4$$

si se supone que también existen no-importa definidos como $D = \sum(9, 12, 14)$.

Solución: Se trata de la misma función empleada en los ejemplos 4.25 a 4.27. En ellos se encontró que la implementación en SOP de costo mínimo es

$$f = x_3x_4 + \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2x_4 + x_1x_4$$

que requiere cuatro compuertas AND, una OR y 13 entradas a ellas, para un costo total de 18.

La implementación en POS de costo mínimo es

$$f = (\overline{x}_3 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + x_3 + \overline{x}_4)$$

que requiere tres compuertas OR, una AND y 11 entradas a ellas, para un costo total de 15.

También puede considerarse una realización multinivel para la función. Al aplicar el concepto de factorización a la expresión en SOP anterior se produce

$$f = (x_1 + \overline{x}_2 + x_3)x_4 + \overline{x}_1\overline{x}_3\overline{x}_4$$

Esta implementación requiere dos compuertas AND, dos OR y 10 entradas a ellas, para un costo total de 14. En comparación con las implementaciones en SOP y POS, ésta tiene el costo más

bajo en términos de compuertas y entradas, pero resulta en un circuito más lento porque existen tres niveles de compuertas por los que las señales deben propagarse. Desde luego, si esta función se implementa en un FPGA, entonces sólo se necesita una LUT.

Ejemplo 4.29 **Problema:** En varios FPGA comerciales los bloques lógicos son LUT de cuatro entradas. Se pueden usar dos de tales LUT, conectadas como se muestra en la figura 4.54, para implementar funciones de siete variables usando la descomposición

$$f(x_1, \ldots, x_7) = f[g(x_1, \ldots, x_4), x_5, x_6, x_7]$$

Es fácil ver que funciones tales como $f = x_1x_2x_3x_4x_5x_6x_7$ y $f = x_1+x_2+x_3+x_4+x_5+x_6+x_7$ pueden implementarse de esta forma. Demuestre que existen otras funciones de siete variables que no pueden implementarse con dos LUT de cuatro entradas.

Solución: La tabla de verdad de una función de siete variables puede ordenarse como se bosqueja en la figura 4.55. Hay $2^7 = 128$ mintérminos. Cada combinación de las variables x_1, x_2, x_3 y x_4 selecciona una de las 16 columnas de la tabla de verdad, mientras que cada combinación de x_5, x_6 y x_7 selecciona una de ocho filas. Puesto que hay que usar el circuito de la figura 4.54, la tabla de verdad para f también puede definirse en términos de la subfunción g. En este caso, g es la que selecciona una de las 16 columnas de la tabla de verdad, en vez de x_1, x_2, x_3 y x_4. Como g sólo puede tener dos valores posibles, 0 y 1, nada más se pueden tener dos columnas en la tabla de verdad. Esto es posible si únicamente existen dos patrones distintos de 1 y 0 en las 16 columnas de la figura 4.54. Por tanto, sólo un subconjunto relativamente pequeño de funciones de siete variables pueden realizarse con sólo dos LUT.

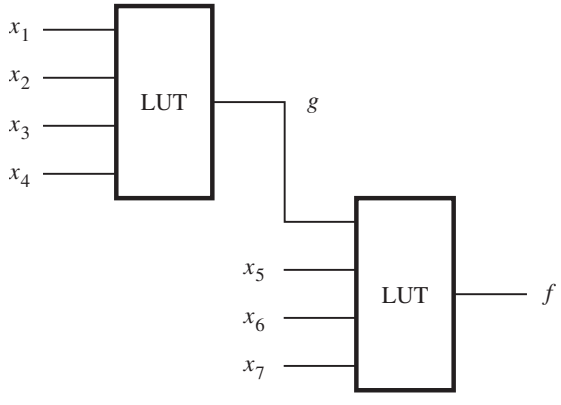


Figura 4.54 Circuito para el ejemplo 4.29.

$x_5x_6x_7$ \ $x_1x_2x_3x_4$	0000	0001	• • •	1110	1111
000	m_0	m_8		m_{112}	m_{120}
001	m_1	m_9		m_{113}	m_{121}
010	m_2	m_{10}		m_{114}	m_{122}
011	m_3	m_{11}	• • •	m_{115}	m_{123}
100	m_4	m_{12}		m_{116}	m_{124}
101	m_5	m_{13}		m_{117}	m_{125}
110	m_6	m_{14}		m_{118}	m_{126}
111	m_7	m_{15}		m_{119}	m_{127}

Figura 4.55 Posible formato para tablas de verdad de funciones de siete variables.

PROBLEMAS

Al final del libro se proporcionan las respuestas a los problemas marcados con asterisco.

***4.1** Encuentre las formas en SOP y POS de costo mínimo para la función $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 5)$.

***4.2** Repita el problema 4.1 para la función $f(x_1, x_2, x_3) = \sum m(1, 4, 7) + D(2, 5)$.

4.3 Repita el problema 4.1 para la función $f(x_1, \ldots, x_4) = \prod M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15)$.

4.4 Repita el problema 4.1 para la función $f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7)$.

***4.5** Repita el problema 4.1 para la función $f(x_1, \ldots, x_5) = \prod M(1, 4, 6, 7, 9, 12, 15, 17, 20, 21, 22, 23, 28, 31)$.

4.6 Repita el problema 4.1 para la función $f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$.

4.7 Repita el problema 4.1 para la función $f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$.

4.8 Encuentre cinco funciones de tres variables para las que la forma en producto de sumas tenga menor costo que la forma en suma de productos.

***4.9** Una función lógica de cuatro variables que es igual a 1 si cualesquiera tres o las cuatro de sus variables son iguales a 1 se llama función *mayoritaria*. Diseñe un circuito en SOP de costo mínimo que implemente esta función mayoritaria.

4.10 Derive una realización en costo mínimo de la función de cuatro variables que es igual a 1 si exactamente dos o exactamente tres de sus variables son iguales a 1; de otro modo es igual a 0.

***4.11** Prueba o muestre un contraejemplo para la afirmación siguiente: si una función f tiene una única expresión en SOP de costo mínimo, entonces también tiene una única expresión en POS de costo mínimo.

***4.12** Un circuito con dos salidas tiene que implementar las funciones siguientes:

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$

$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Diseñe el circuito de costo mínimo y compare su costo con los costos combinados de dos circuitos que implementen f y g por separado. Suponga que las variables de entrada están disponibles tanto en forma sin complementar como complementada.

4.13 Repita el problema 4.12 para las funciones siguientes

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$

$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

***4.14** Implemente el circuito lógico de la figura 4.23 usando solamente compuertas NAND.

***4.15** Implemente el circuito lógico de la figura 4.23 usando solamente compuertas NOR.

4.16 Implemente el circuito lógico de la figura 4.25 usando solamente compuertas NAND.

4.17 Implemente el circuito lógico de la figura 4.25 usando solamente compuertas NOR.

***4.18** Considere la función $f = x_3x_5 + \bar{x}_1x_2x_4 + x_1\bar{x}_2\bar{x}_4 + x_1x_3\bar{x}_4 + \bar{x}_1x_3x_4 + \bar{x}_1x_2x_5 + x_1\bar{x}_2x_5$. Derive un circuito de costo mínimo que la implemente usando compuertas NOT, AND y OR.

4.19 Derive un circuito de costo mínimo que implemente la función $f(x_1, \ldots, x_4) = \sum m(4, 7, 8, 11) + D(12, 15)$.

4.20 Encuentre la realización más simple de la función $f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 7, 9, 10, 13, 14)$, si se supone que las compuertas lógicas tienen una entrada de carga máxima de dos.

***4.21** Encuentre el circuito de costo mínimo para la función $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 13, 14, 15)$. Suponga que las variables de entrada están disponibles sólo en forma sin complementar. (*Sugerencia*: aplique descomposición funcional.)

4.22 Use descomposición funcional para encontrar la mejor implementación de la función $f(x_1, \ldots, x_5) = \sum m(1, 2, 7, 9, 10, 18, 19, 25, 31) + D(0, 15, 20, 26)$. ¿Cómo se compara su implementación con la implementación en SOP de costo más bajo? Proporcione los costos.

***4.23** Use el método tabular expuesto en la sección 4.9 para hallar una realización en SOP de costo mínimo para la función

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$$

4.24 Repita el problema 4.23 para la función

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 6, 8, 9, 15) + D(3, 7, 11, 13)$$

4.25 Repita el problema 4.23 para la función

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 5, 7, 9, 11) + D(8, 12, 13, 14)$$

4.26 Demuestre que son válidas las reglas parecidas a distributiva siguientes

$$(A \cdot B)\#C = (A\#C) \cdot (B\#C)$$
$$(A + B)\#C = (A\#C) + (B\#C)$$

4.27 Use la representación cúbica y el método expuesto en la sección 4.10 para hallar una realización en SOP de costo mínimo de la función $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$.

4.28 Repita el problema 4.27 para la función $f(x_1, \ldots, x_5) = \overline{x}_1\overline{x}_3\overline{x}_5 + x_1x_2\overline{x}_3 + x_2x_3\overline{x}_4x_5 + x_1\overline{x}_2\overline{x}_3x_4 + x_1x_2x_3x_4\overline{x}_5 + \overline{x}_1x_2x_4\overline{x}_5 + \overline{x}_1\overline{x}_3x_4x_5$.

4.29 Utilice la representación cúbica y el método expuesto en la sección 4.10 para hallar una realización en SOP de costo mínimo de la función $f(x_1, \ldots, x_4)$, definida por el conjunto ON ON = {00x0, 100x, x010, 1111} y el conjunto de no-importa DC = {00x1, 011x}.

4.30 En la sección 4.10.1 mostramos cómo usar la operación $*$ para hallar los implicantes primos de una función *f*. Otra posibilidad consiste en encontrar los implicantes primos mediante la expansión de los implicantes en la cobertura inicial de la función. Un implicante se *expande* eliminando una literal para crear un implicante más grande (en términos del número de vértices cubiertos). Un implicante más grande sólo es válido si no incluye vértice alguno para el que $f = 0$. Los implicantes válidos más grandes que se obtienen en el proceso de expansión son los primos. En la figura P4.1 se ilustra la expansión del implicante $\overline{x}_1x_2x_3$ de la función de la figura 4.9, que también se usa en el ejemplo 4.16. A partir de la figura 4.9, note que

$$\overline{f} = x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

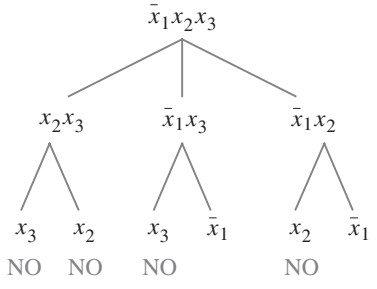


Figura P4.1 Expansión del implicante $\overline{x}_1x_2x_3$.

En la figura P4.1 la palabra NO se usa para indicar que el término expandido no es válido porque incluye uno o más vértices de $\overline{f}$. A partir de la gráfica es claro que los implicantes válidos más grandes que surgen de esta expansión son x_2x_3 y $\overline{x}_1$; se trata de los implicantes primos de *f*.

Expanda los otros cuatro implicantes dados en la cobertura inicial del ejemplo 4.14 para encontrar todos los implicantes primos de *f*. ¿Cuál es la complejidad relativa de este procedimiento en comparación con la técnica del producto $*$?

4.31 Repita el problema 4.30 para la función del ejemplo 4.17. Expanda los implicantes dados en la cobertura inicial C^0.

***4.32** Considere las expresiones lógicas

$$f = x_1\bar{x}_2\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5 + x_1x_2x_4x_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2x_3x_5 + \bar{x}_2\bar{x}_3x_4\bar{x}_5 + x_1x_2x_3x_4\bar{x}_5$$
$$g = \bar{x}_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_3x_4\bar{x}_5 + x_1\bar{x}_2x_4\bar{x}_5 + x_1x_3x_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3x_4x_5$$

Pruebe o debata que $f = g$.

4.33 Considere el circuito de la figura P4.2, que implementa las funciones f y g. ¿Cuál es su costo, si se supone que las variables de entrada están disponibles tanto en verdadero como en complementado? Rediseñe el circuito para implementar las mismas funciones, pero a un costo cuan bajo sea posible. ¿Cuál es el costo de este circuito?

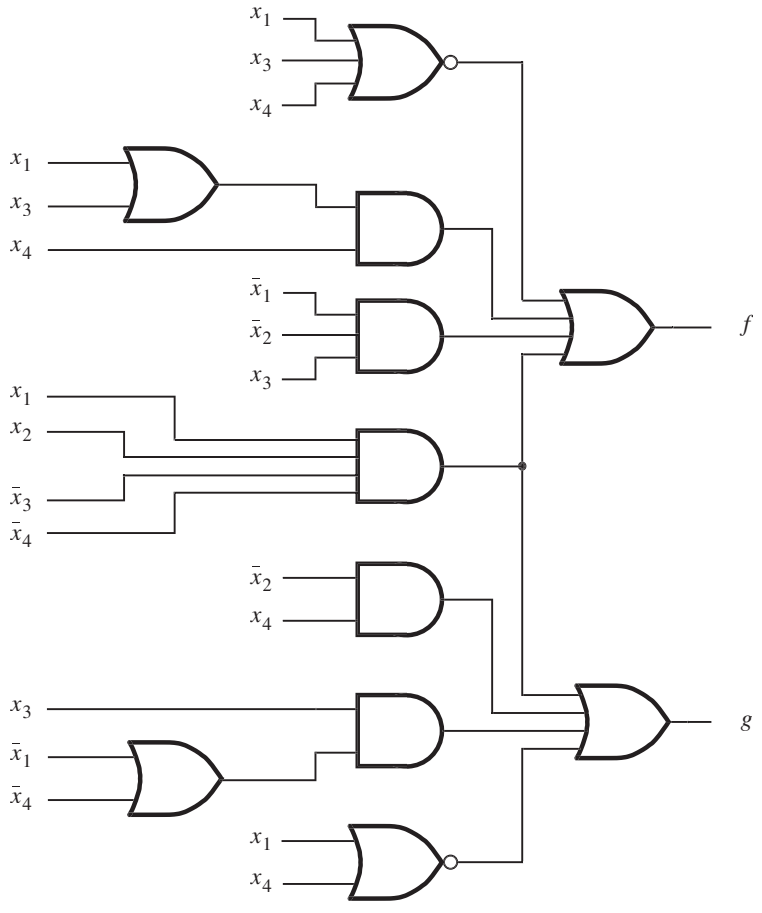


Figura P4.2 Circuito para el problema 4.33.

4.34 Repita el problema 4.33 para el circuito de la figura P4.3. En el circuito use sólo compuertas NAND.

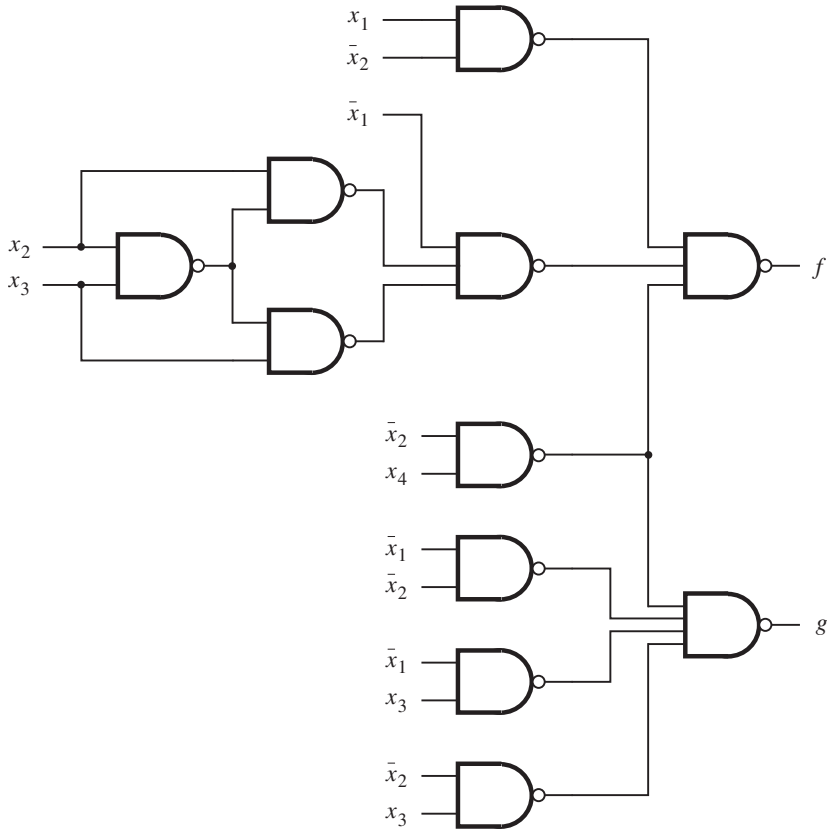


Figura P4.3 Circuito para el problema 4.34.

4.35 Escriba el código de VHDL para implementar el circuito de la figura 4.25*b*.

4.36 Escriba el código de VHDL para implementar el circuito de la figura 4.27*c*.

4.37 Escriba el código de VHDL para implementar el circuito de la figura 4.28*b*.

4.38 Escriba el código de VHDL para implementar la función $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15)$.

4.39 Escriba el código de VHDL para implementar la función $f(x_1, \ldots, x_4) = \sum m(1, 4, 7, 14, 15) + D(0, 5, 9)$.

4.40 Escriba el código de VHDL para implementar la función $f(x_1, \ldots, x_4) = \Pi M(6, 8, 9, 12, 13)$.

4.41 Escriba el código de VHDL para implementar la función $f(x_1, \ldots, x_4) = \Pi M(3, 11, 14) + D(0, 2, 10, 12)$.

BIBLIOGRAFÍA

1. M. Karnaugh, "A Map Method for Synthesis of Combinatorial Logic Circuits", *Transactions of AIEE, Communications and Electronics* 72, parte 1, noviembre de 1953, pp. 593-599.
2. R. L. Ashenhurst, "The Decomposition of Switching Functions", Proc. of the Symposium on the Theory of Switching, 1957, *Vol. 29 of Annals of Computation Laboratory* (Harvard University: Cambridge, MA, 1959), pp. 74-116.
3. F. J. Hill y G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4a. ed. (Wiley: Nueva York, 1993).
4. T. Sasao, *Logic Synthesis and Optimization* (Kluwer: Boston, MA, 1993).
5. S. Devadas, A. Gosh y K. Keutzer, *Logic Synthesis* (McGraw-Hill: Nueva York, 1994).
6. W. V. Quine, "The Problem of Simplifying Truth Functions", *Amer. Math. Monthly* 59 (1952), pp. 521-531.
7. E. J. McCluskey Jr., "Minimization of Boolean Functions", *Bell System Tech. Journal*, noviembre de 1956, pp. 1417-1444.
8. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. J. F. Wakerly, *Digital Design Principles and Practices*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
10. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
11. C. H. Roth Jr., *Fundamentals of Logic Design*, 4a. ed. (West: St. Paul, MN, 1993).
12. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
13. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
14. J. P. Daniels, *Digital Design from Zero to One* (Wiley: Nueva York, 1996).
15. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
16. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
17. M. M. Mano, *Digital Design*, 3a. ed. (Prentice-Hall: Upper Saddle River, NJ, 2001).

18. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

19. R. K. Brayton, G. D. Hachtel, C. T. McMullen y A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer: Boston, MA, 1984).

20. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli y A. R. Wang, "MIS: A Multiple-Level Logic Synthesis Optimization System", *IEEE Transactions on Computer-Aided Design*, CAD-6, noviembre de 1987, pp. 1062-1081.

21. E. M. Sentovic, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton y A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis", Reporte técnico UCB/ERL M92/41, Laboratorio de Investigación en Electrónica, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California.

22. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill: Nueva York, 1994).

23. N. Sherwani, *Algorithms for VLSI Physical Design Automation* (Kluwer: Boston, MA, 1995).

24. B. Preas y M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings: Redwood City, CA, 1988).

capítulo

5

Representación de números y circuitos aritméticos

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- Cómo se representan los números en las computadoras
- Los circuitos utilizados para realizar operaciones aritméticas
- Problemas de rendimiento en los circuitos grandes
- Cómo usar VHDL para especificar circuitos aritméticos

En este capítulo estudiaremos los circuitos lógicos que realizan operaciones aritméticas. Explicaremos cómo pueden sumarse, restarse y multiplicarse números. También mostraremos cómo escribir código de VHDL para describir los circuitos aritméticos, los cuales ofrecen una plataforma estupenda para ilustrar el poder y la versatilidad de ese lenguaje para especificar ensambles complejos de circuitos lógicos. Los conceptos relativos al diseño de los circuitos aritméticos se aplican con facilidad en muchos otros tipos de circuitos.

Antes de abordar el diseño de los circuitos aritméticos es preciso analizar cómo se representan los números en los sistemas digitales. En los capítulos anteriores explicamos las variables lógicas de un modo general, empleándolas para representar el estado de un interruptor o ciertas condiciones generales. Ahora las utilizaremos para representar números. Se necesita más de una variable para indicar un número, y cada variable corresponde a un dígito de éste.

5.1 REPRESENTACIÓN NUMÉRICA POSICIONAL

Cuando se estudian los números y las operaciones aritméticas es útil usar símbolos estándar. Por ende, para representar la suma ocupamos el símbolo más (+) y para la resta el símbolo menos (−). En capítulos anteriores empleamos + para representar la operación lógica OR y − para denotar la eliminación de un elemento de un conjunto. Ahora usaremos los mismos símbolos para dos propósitos diferentes, mas el significado de cada uno será sin duda claro a partir del contexto de la exposición. En los casos donde pueda haber cierta ambigüedad, explicaremos su significado.

5.1.1 ENTEROS SIN SIGNO

Los números más simples son los enteros. Empezaremos por considerar los enteros positivos y luego incluiremos los negativos. Los números positivos se llaman también *sin signo*, y los que pueden ser negativos se denominan *con signo*. La representación de los números que incluyen un punto en la base (números reales) se explica más adelante.

En el sistema decimal un número consta de dígitos que tienen 10 valores posibles, de 0 a 9, y cada dígito representa un múltiplo de una potencia de 10. Por ejemplo, el número 8547 representa $8 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$. No es común escribir las potencias de 10 con el número, ya que están implícitas en las posiciones de los dígitos. En general, un entero decimal se expresa por medio de una *n-tupla* que comprende *n* dígitos decimales

$$D = d_{n-1}d_{n-2}\cdots d_1d_0$$

que representa el valor

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

Esto se conoce como *representación numérica posicional*.

Puesto que los dígitos tienen 10 posibles valores y cada dígito se evalúa como una potencia de 10, se dice que los números decimales son números de *base* 10, o *raíz* 10. Los números decimales son conocidos, convenientes y fáciles de entender. Sin embargo, en los circuitos digitales no resulta práctico usar dígitos que pueden adquirir 10 valores. En los sistemas digitales se usa el

sistema binario, o de *base* 2, en el que los dígitos pueden ser 0 o 1. Cada dígito binario se llama *bit*. En el sistema numérico binario se emplea la misma representación numérica posicional, de modo que

$$B = b_{n-1}b_{n-2}\cdots b_1b_0$$

representa un entero que tiene el valor

$$\begin{aligned} V(B) &= b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{n-1} b_i \times 2^i \end{aligned} \qquad \textbf{[5.1]}$$

Por ejemplo, el número binario 1101 representa el valor

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Puesto que el patrón particular de un dígito tiene diferentes significados para distintas bases numéricas, las bases (o raíces) se indican con un subíndice cuando pueda haber alguna confusión. Por tanto, para indicar que 1101 es un número en base 2 se escribe $(1101)_2$. Al evaluar la expresión anterior para V se obtiene $V = 8 + 4 + 1 = 13$. Por tanto,

$$(1101)_2 = (13)_{10}$$

Nótese que el intervalo de enteros que puede representarse mediante un número binario depende del número de bits utilizados. Por ejemplo, con cuatro bits el número más grande es $(1111)_2 = (15)_{10}$. Un ejemplo de un número más grande es $(10110111)_2 = (183)_{10}$. En general, el uso de n bits permite representar enteros en el intervalo de 0 a $2^n - 1$.

En un número binario el bit del extremo derecho se denomina *bit menos significativo* (LSB, *least-significant bit*). El bit del extremo izquierdo en un entero sin signo, que tiene asociada la potencia más alta de 2, se llama *bit más significativo* (MSB, *most-significant bit*). En los sistemas digitales es útil considerar varios bits juntos como un grupo. Un grupo de cuatro bits se llama *nibble* y uno de ocho bits se denomina *byte*.

5.1.2 CONVERSIÓN ENTRE SISTEMAS DECIMAL Y BINARIO

Un número binario se convierte en un número decimal simplemente aplicando la ecuación 5.1 y evaluándolo con aritmética decimal. La conversión de un número decimal en uno binario no es tan directa. Puede llevarse a cabo mediante la división sucesiva entre 2 del número decimal, del modo siguiente. Supóngase que un número decimal $D = d_{k-1}\cdots d_1d_0$, con un valor V, se convertirá en un número binario $B = b_{n-1}\cdots b_2b_1b_0$. Por tanto,

$$V = b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0$$

Si V se divide entre 2 el resultado es

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

El cociente de esta división entera es $b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2 + b_1$, y el residuo es b_0. Si el residuo es 0, entonces $b_0 = 0$; si es 1, entonces $b_0 = 1$. Obsérvese que el cociente es justo otro número binario, que comprende $n - 1$ bits, en lugar de n bits. Al dividir este número entre 2 queda el residuo b_1. El nuevo cociente es

$$b_{n-1} \times 2^{n-3} + \cdots + b_2$$

Si se continúa el proceso de dividir el nuevo cociente entre 2 y se determina un bit en cada paso se producirán todos los bits del número binario. El proceso continúa hasta que el cociente es 0. En la figura 5.1 se ilustra el proceso de conversión mediante el ejemplo $(857)_{10} = (1101011001)_2$. Nótese que primero se genera el bit menos significativo (LSB) y al final queda el más significativo (MSB).

5.1.3 REPRESENTACIONES OCTAL Y HEXADECIMAL

La representación numérica posicional puede usarse para cualquier base. Si ésta es r, entonces el número

$$K = k_{n-1}k_{n-2} \cdots k_1k_0$$

tiene el valor

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

En el texto, el interés se limita a las bases más prácticas. Usaremos números decimales porque son los que utiliza la gente, y emplearemos números binarios porque son los que ocupan las computadoras. Además, son útiles otras dos bases: 8 y 16. Los números representados con la

Convertir $(857)_{10}$

			Residuo	
857 ÷ 2	=	428	1	LSB
428 ÷ 2	=	214	0	
214 ÷ 2	=	107	0	
107 ÷ 2	=	53	1	
53 ÷ 2	=	26	1	
26 ÷ 2	=	13	0	
13 ÷ 2	=	6	1	
6 ÷ 2	=	3	0	
3 ÷ 2	=	1	1	
1 ÷ 2	=	0	1	MSB

El resultado es $(1101011001)_2$

Figura 5.1 Conversión de decimal a binario.

base 8 se llaman números *octales* y los de base 16 se denominan *hexadecimales*. En la representación octal, los valores de los dígitos van de 0 a 7; en la hexadecimal (que se abrevia *hexa*), cada dígito puede tener uno de 16 valores. Los primeros 10 se denotan igual que en el sistema decimal: de 0 a 9. Los dígitos que corresponden a los valores decimales 10, 11, 12, 13, 14 y 15 se denotan mediante las letras A, B, C, D, E y F. En la tabla 5.1 se presentan los primeros 18 enteros en estos sistemas numéricos.

Tabla 5.1 Números en diferentes sistemas.

Decimal	Binario	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

El sistema numérico dominante en las computadoras es el binario. Los sistemas octal y hexadecimal se usan porque brindan una notación abreviada para los números binarios. Un dígito octal representa tres bits. Por tanto, un número binario se convierte en un número octal tomando grupos de tres bits, desde el menos significativo, y sustituyéndolos con el correspondiente dígito octal. Por ejemplo, 101011010111 se convierte en

$$\underbrace{1\,0\,1}_{5} \qquad \underbrace{0\,1\,1}_{3} \qquad \underbrace{0\,1\,0}_{2} \qquad \underbrace{1\,1\,1}_{7}$$

lo que significa que $(101011010111)_2 = (5327)_8$. Si el número de bits no es un múltiplo de tres, entonces se agrega 0 a la izquierda del bit más significativo. Por ejemplo, $(10111011)_2 = (273)_8$ porque

$$\underbrace{0\,1\,0}_{2} \qquad \underbrace{1\,1\,1}_{7} \qquad \underbrace{0\,1\,1}_{3}$$

La conversión de octal a binario es así de directa; cada dígito octal simplemente se sustituye por tres bits que denotan el mismo valor.

De manera similar, un dígito hexadecimal se representa con cuatro bits. Por ejemplo, un número de 16 bits se representa con cuatro dígitos hexa, como en

$$(1010111100100101)_2 = (\text{AF25})_{16}$$

porque

$$\underbrace{1\,0\,1\,0}_{\text{A}} \qquad \underbrace{1\,1\,1\,1}_{\text{F}} \qquad \underbrace{0\,0\,1\,0}_{2} \qquad \underbrace{0\,1\,0\,1}_{5}$$

Si el número de bits no es múltiplo de cuatro, se agregan ceros a la izquierda del bit más significativo. Por ejemplo, $(1101101000)_2 = (368)_{16}$, porque

$$\underbrace{0\,0\,1\,1}_{3} \qquad \underbrace{0\,1\,1\,0}_{6} \qquad \underbrace{1\,0\,0\,0}_{8}$$

La conversión de hexadecimal a binario involucra la sustitución directa de cada dígito hexa por cuatro bits que denotan el mismo valor.

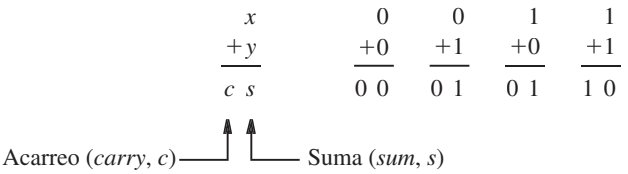
Los números binarios usados en las computadoras modernas con frecuencia tienen 32 o 64 bits. Escritos como *n*-tuplas binarias (a veces llamadas vectores bit), tales números son complicados para que la gente los manipule. Es mucho más simple lidiar con ellos en la forma de números de 8 o 16 dígitos. Puesto que las operaciones aritméticas en un sistema digital suelen comprender números binarios, nos centraremos en los circuitos que los usan. En ocasiones emplearemos la representación hexadecimal como una cómoda descripción abreviada.

Ya expusimos los números más simples: los enteros sin signo. Es necesario ser capaz de trabajar con varios tipos de números. Más adelante, en este capítulo, se tratará la representación de los números con signo, los números con punto fijo y con punto flotante. Pero antes examinaremos algunos circuitos simples que operan sobre los números para dar al lector una idea de los circuitos digitales que realizan operaciones aritméticas y alentarlo así para la discusión posterior.

5.2 SUMA DE NÚMEROS SIN SIGNO

La suma binaria se realiza igual que la decimal, excepto que los valores de los dígitos individuales sólo pueden ser 0 o 1. La suma de dos números de un bit conlleva cuatro posibles combinaciones, como se indica en la figura 5.2*a*. Se necesitan dos bits para representar el resultado de la suma. El bit del extremo derecho se llama *suma* (*sum*), *s*. El del extremo izquierdo, que se produce como acarreo cuando los dos bits que se suman son iguales a 1, se denomina *acarreo* (*carry*), *c*. La operación suma se define en forma de una tabla de verdad en el inciso *b*) de la figura. El bit suma *s* es la función XOR, expuesta en la sección 3.9.1. El acarreo *c* es la función AND de las entradas *x* y *y*. En la figura 5.2*c* se muestra una realización del circuito de estas funciones. Este circuito, que implementa la suma sólo de dos bits, se llama medio sumador (HA, *half-adder*).

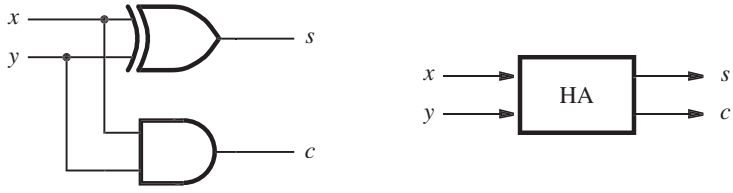
Un caso más interesante es cuando se involucran números más grandes que tienen varios bits. Aun así es necesario sumar cada par de bits, pero, para cada posición de bit *i*, la operación suma puede incluir un *acarreo* desde la posición de bit $i - 1$.



a) Los cuatro casos posibles

		Acarreo	Suma
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

b) Tabla de verdad



c) Circuito *d*) Símbolo gráfico

Figura 5.2 Medio sumador.

En la figura 5.3 se presenta un ejemplo de la operación suma. Los dos operandos son $X = (01111)_2 = (15)_{10}$ y $Y = (01010)_2 = (10)_{10}$. Nótese que se usan cinco bits para representar X y Y. Con cinco bits es posible representar enteros que estén en el intervalo de 0 a 31; por ende, la suma $S = X + Y = (25)_{10}$ también puede denotarse como un entero de cinco bits. Obsérvese también la etiqueta de cada bit, tal que $X = x_4x_3x_2x_1x_0$ y $Y = y_4y_3y_2y_1y_0$. En la figura se muestran

$X = x_4x_3x_2x_1x_0$	0 1 1 1 1	$(15)_{10}$
$+Y = y_4y_3y_2y_1y_0$	0 1 0 1 0	$(10)_{10}$
	1 1 1 0 ← Acarreos generados	
$S = s_4s_3s_2s_1s_0$	1 1 0 0 1	$(25)_{10}$

Figura 5.3 Ejemplo de suma.

los acarreos generados durante el proceso de suma. Por ejemplo, un acarreo de 0 se genera cuando se suman x_0 y y_0; un acarreo de 1 se produce cuando se suman x_1 y y_1, etcétera.

En los capítulos 2 y 4 diseñamos circuitos lógicos especificando primero sus comportamientos en la forma de tabla de verdad. Este enfoque no es práctico al diseñar un circuito sumador que puede sumar los números de cinco bits de la figura 5.3. La tabla de verdad requerida tendría 10 variables de entrada, cinco para cada número X y Y. ¡Tendría $2^{10} = 1024$ filas! Un enfoque mejor consiste en considerar la suma de cada par de bits, x_i y y_i por separado.

Para la posición de bit 0 no hay acarreo y, por tanto, la suma es la misma que para la figura 5.2. Para cada otra posición de bit i, la suma comprende los bits x_i y y_i, y un acarreo en c_i. Las funciones suma y acarreo de las variables x_i, y_i y c_i se especifican en la tabla de verdad de la figura 5.4*a*. El bit suma, s_i, es la suma módulo 2 de x_i, y_i y c_i. El acarreo, c_{i+1}, es igual a 1 si la suma de x_i, y_i y c_i es igual a 2 o a 3. Los mapas de Karnaugh para estas funciones se muestran en el inciso *b*) de la figura. Para la función acarreo la realización óptima en suma de productos es

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Para la función s_i, una realización en suma de productos es

$$s_i = \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + \overline{x}_i \overline{y}_i c_i + x_i y_i c_i$$

Una forma más atractiva de implementar esta función es con compuertas XOR, como se explica a continuación.

Uso de compuertas XOR

La función XOR de dos variables se define como $x_1 \oplus x_2 = \overline{x}_1 x_2 + x_1 \overline{x}_2$. La expresión anterior para el bit suma puede manipularse del modo siguiente en una forma que sólo use operaciones XOR

$$\begin{aligned} s_i &= (\overline{x}_i y_i + x_i \overline{y}_i)\overline{c}_i + (\overline{x}_i \overline{y}_i + x_i y_i) c_i \\ &= (x_i \oplus y_i)\overline{c}_i + \overline{(x_i \oplus y_i)} c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

La operación XOR es asociativa; por tanto, puede escribirse

$$s_i = x_i \oplus y_i \oplus c_i$$

En consecuencia, para realizar s_i es posible utilizar una sola compuerta XOR de tres entradas.

La compuerta XOR genera como salida una suma módulo 2 de sus entradas. La salida es igual a 1 si un número impar de entradas tiene el valor 1; de otro modo es igual a 0. Por ello, la XOR a veces se llama *función impar*. Obsérvese que la XOR no tiene minitérminos que puedan combinarse en un término producto más grande, como es evidente a partir del patrón de marcas de la función s_i en el mapa de la figura 5.4*b*. En la figura 5.4*c* se proporciona el circuito lógico que implementa la tabla de verdad de la figura 5.4*a*. Este circuito se conoce como *sumador completo* (FA, *full-adder*).

Otra característica interesante de las compuertas XOR es que puede considerarse que una compuerta XOR de dos entradas utiliza una entrada como señal de control que determina si a través de la compuerta pasará como valor de salida el valor verdadero o complementado de la otra entrada. Esto es claro a partir de la definición de XOR, donde $x_i \oplus y_i = \overline{x}y + x\overline{y}$. Sea x la entrada de control. Entonces, si $x = 0$, la salida será igual al valor de y. Pero si $x = 1$, la salida

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

a) Tabla de verdad

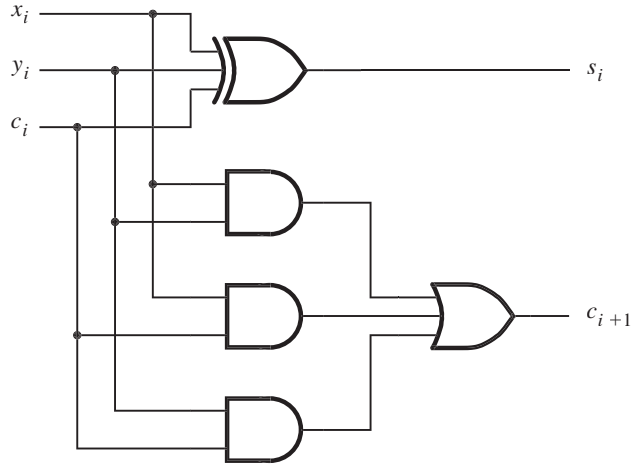
c_i \ $x_i y_i$	00	01	11	10
0		1		1
1	1		1	

$s_i = x_i \oplus y_i \oplus c_i$

c_i \ $x_i y_i$	00	01	11	10
0			1	
1		1	1	1

$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

b) Mapas de Karnaugh



c) Circuito

Figura 5.4 Sumador completo.

será igual al complemento de y. En la deducción anterior usamos manipulación algebraica para derivar $s_i = (x_i \oplus y_i) \oplus c_i$. Pudimos haber obtenido la misma expresión de inmediato haciendo la observación siguiente. En la mitad superior de la tabla de verdad de la figura 5.4*a*, c_i es igual a 0, y la función suma s_i es la XOR de x_i y y_i. En la mitad inferior de la tabla, c_i es igual a 1, mientras que s_i es la versión complementada de su mitad superior. Esta observación conduce directamente a nuestra expresión que utiliza dos operaciones XOR de dos entradas. En la

sección 5.3.3 encontraremos un importante ejemplo del uso de las compuertas XOR para pasar señales verdaderas o complementadas bajo el control de otra señal.

En la explicación precedente vimos el complemento de la operación XOR, que se denotó como $\overline{x \oplus y}$. Esta operación se usa de manera tan común que se le ha dado un nombre distintivo: XNOR. Con frecuencia se utiliza el símbolo especial $\odot$, para denotar la operación XNOR:

$$x \odot y = \overline{x \oplus y}$$

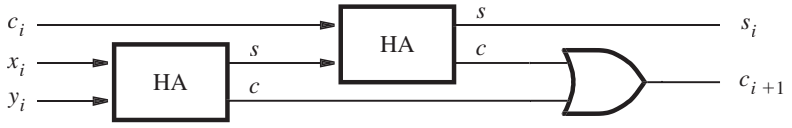
La operación XNOR a veces también se conoce como *operación coincidencia* porque produce la salida de 1 cuando sus entradas coinciden en valor; es decir, ambas son 0 o ambas son 1.

5.2.1 SUMADOR COMPLETO DESCOMPUESTO

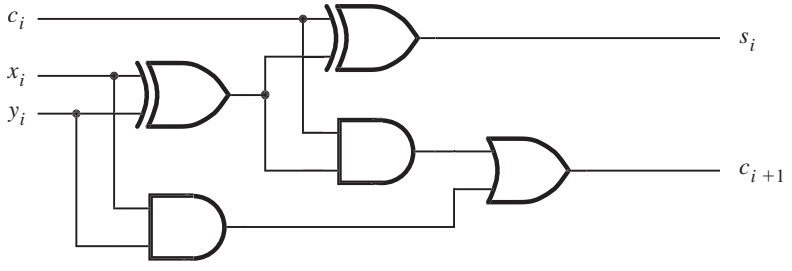
En vista de los nombres utilizados para los circuitos, cabe esperar que un sumador completo se construya con medios sumadores. Esto puede lograrse creando un circuito multinivel del tipo expuesto en la sección 4.6.2. El circuito se presenta en la figura 5.5. Emplea dos medios sumadores para formar un sumador completo. Se deja como ejercicio al lector comprobar su exactitud.

5.2.2 SUMADOR CON ACARREO EN CASCADA

Para realizar la adición a mano, se comienza desde el dígito menos significativo y se agregan pares de dígitos; luego se continúa hacia el dígito más significativo. Si en la posición *i* se produce



a) Diagrama de bloques



b) Diagrama detallado

Figura 5.5 Implementación descompuesta del circuito sumador completo.

un acarreo, éste se suma a los operandos en la posición $i + 1$. El mismo ordenamiento se usa en un circuito lógico que realice sumas. Para cada posición de bit puede utilizarse un circuito sumador completo, conectado como se muestra en la figura 5.6. Nótese que para ser consistentes con la forma habitual de escribir números, la posición del bit menos significativo está a la derecha. Los acarreos producidos por los sumadores completos se propagan a la izquierda.

Cuando los operandos X y Y se aplican como entradas al sumador, se requiere cierto tiempo antes de que la salida suma (*sum*), S, sea válida. Cada sumador completo introduce cierto retraso antes de que sus salidas s_i y c_{i+1} sean válidas. Denotemos tal retraso como Δt. Por tanto, el acarreo de la primera etapa, c_1, llega a la segunda etapa Δt después de la aplicación de las entradas x_0 y y_0. El acarreo de la segunda etapa, c_2, llega a la tercera con un retraso $2\Delta t$, etc. La señal c_{n-1} es válida después de un retraso de $(n - 1)\Delta t$, lo que significa que la suma completa está disponible después de un retraso de $n\Delta t$. Por la forma en que las señales "se propagan" a través de las etapas del sumador completo, el circuito de la figura 5.6 se denomina *sumador con acarreo en cascada*.

El retraso en que se incurre para producir la suma final y el acarreo en un sumador de acarreo en cascada depende del tamaño de los números. Cuando se usan números de 32 o 64 bits, este retraso puede llegar a ser inaceptablemente alto. Puesto que el circuito en cada sumador completo deja poco espacio para una reducción drástica del retraso, quizá sea necesario buscar diferentes estructuras para implementar sumadores de n bits. En la sección 5.4 expondremos una técnica para construir sumadores de alta velocidad.

Hasta el momento hemos trabajado sólo con enteros sin signo. La suma de tales números no requiere un acarreo para la etapa 0. En la figura 5.6 incluimos c_0 en el diagrama, de modo que el sumador con acarreo en cascada también sirva para restar números, como veremos en la sección 5.3.

5.2.3 EJEMPLO DE DISEÑO

Supóngase que se necesita un circuito que multiplique por 3 un número sin signo de ocho bits. Sea $A = a_7a_6 \ldots a_1a_0$ el número y $P = p_9p_8 \ldots p_1p_0$ el producto $P = 3A$. Nótese que se necesitan 10 bits para representar el producto.

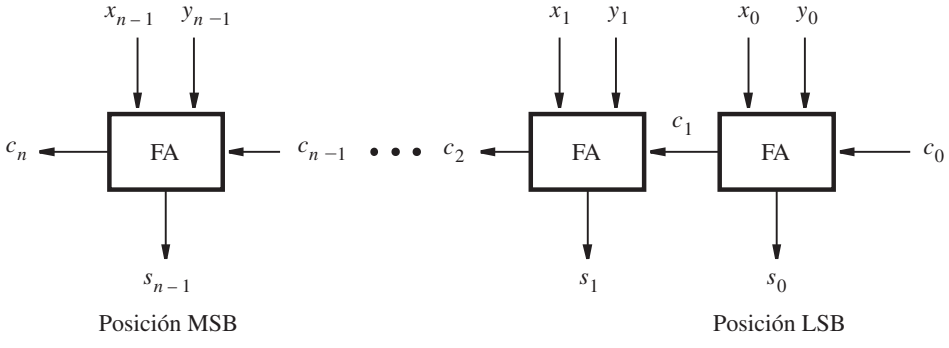


Figura 5.6 Sumador de n-bits con acarreo en cascada.

Un enfoque simple para diseñar ese circuito consiste en usar dos sumadores con acarreo en cascada para sumar tres copias del número A, como se ilustra en la figura 5.7*a*. El símbolo que denota a cada sumador es uno utilizado comúnmente para los sumadores. Las letras x_i, y_i, s_i y c_i indican el significado de las entradas y las salidas de acuerdo con la figura 5.6. El primer sumador produce $A + A = 2A$. Su resultado se representa como ocho bits suma y el acarreo proveniente del bit más significativo. El segundo sumador produce $2A + A = 3A$. Tiene que ser un sumador de nueve bits para poder manejar los nueve bits de $2A$, que el primer sumador genera. Puesto que las entradas y_i deben dirigirse sólo por los ocho bits de A, la novena entrada y_8 se conecta a una constante 0.

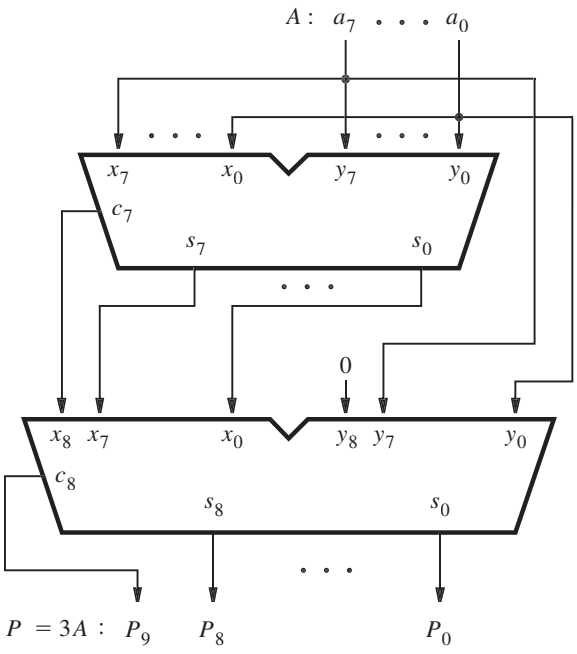
Este método es directo, pero poco eficiente. Puesto que $3A = 2A + A$, se observa que $2A$ puede generarse corriendo los bits de A una posición de un bit a la izquierda, lo que produce el patrón de bits $a_7a_6a_5a_4a_3a_2a_1a_0$0. De acuerdo con la ecuación 5.1, este patrón es igual a $2A$. Entonces basta un solo sumador con acarreo en cascada para implementar $3A$, como se muestra en la figura 5.7*b*. Éste es en esencia el mismo circuito que el segundo sumador del inciso *a*) de la figura. Nótese que la entrada x_0 se conecta a una constante 0. Nótese también que en el segundo sumador del inciso *a*), el valor de x_0 siempre es 0, aun cuando lo dirija el bit menos significativo, s_0, de la suma del primer sumador. Como $x_0 = y_0 = a_0$ en el primer sumador, el bit suma s_0 será 0, ya sea que a_0 sea 0 o 1.

5.3 NÚMEROS CON SIGNO

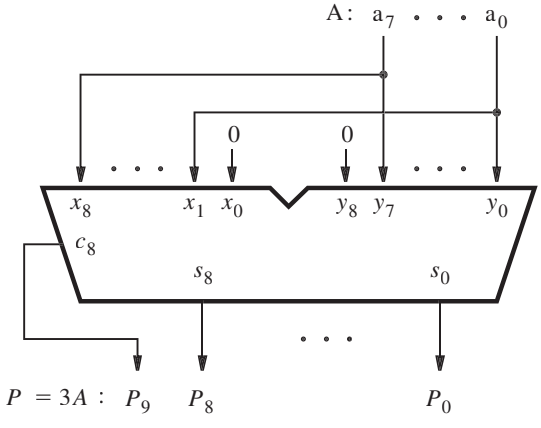
En el sistema decimal el signo de un número se indica mediante los símbolos + o − a la izquierda del dígito más significativo. En el sistema binario el *signo* de un número se denota por el bit del extremo izquierdo. Para un número positivo ese bit es igual a 0, y para un número negativo es igual a 1. Por tanto, en los números con signo el bit del extremo izquierdo representa el signo y los restantes $n - 1$ bits representan la magnitud, como se ilustra en la figura 5.8. Es importante notar la diferencia en la ubicación del bit más significativo (MSB). En los números sin signo todos los bits representan la magnitud de un número; por ende, los n bits son *significativos* al definir la magnitud. En consecuencia, el MSB es el bit más a la izquierda, b_{n-1}. En los números con signo existen $n - 1$ bits significativos, y el MSB se halla en la posición del bit b_{n-2}.

5.3.1 NÚMEROS NEGATIVOS

Los números positivos se indican con la representación numérica posicional, como explicamos en la sección anterior. Los números negativos pueden representarse de tres formas: signo y magnitud, complemento a 1 y complemento a 2.



a) Enfoque poco eficiente



b) Diseño eficiente

Figura 5.7 Circuito que multiplica por 3 un número sin signo de ocho bits.

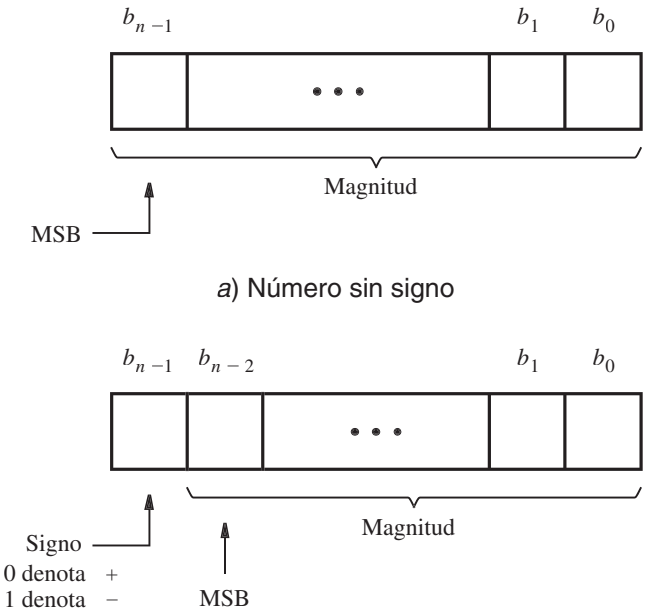


Figura 5.8 Formatos para la representación de enteros.

Representación signo y magnitud

En la representación decimal, la magnitud de los números positivos y negativos se expresa igual. El signo distingue un número como positivo o negativo. Este esquema se llama representación numérica *signo y magnitud*. El mismo esquema puede usarse con números binarios, caso en el que el bit del signo es 0 o 1 para los números positivos o negativos, respectivamente. Por ejemplo, si utilizamos números de cuatro bits, entonces $+5 = 0101$ y $-5 = 1101$. Por su similitud con los números decimales de signo y magnitud, esta representación es fácil de entender. Sin embargo, como muy pronto veremos, no está bien adaptada para usarla en las computadoras. Las representaciones más adecuadas se basan en sistemas de complementación, como explicamos enseguida.

Representación en complemento a 1

En un sistema numérico complementario, los números negativos se definen de acuerdo con una operación de sustracción que implica números positivos. Consideraremos dos esquemas para números binarios: el complemento a 1 y el complemento a 2. En el esquema de *complemento a 1*, un número negativo de n bits, K, se obtiene restando su equivalente número positivo, P, de $2^n - 1$; es decir, $K = (2^n - 1) - P$. Por ejemplo, si $n = 4$, entonces $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$. Si $+5$ se convierte en negativo se obtiene $-5 = 1111 - 0101 = 1010$.

De manera similar, $+3 = 0011$ y $-3 = 1111 - 0011 = 1100$. Es claro que el complemento a 1 puede obtenerse simplemente complementando cada bit del número, incluido el bit del signo. Si bien los números en complemento a 1 son fáciles de derivar, tienen ciertos inconvenientes cuando se emplean en operaciones aritméticas, como veremos en la sección siguiente.

Representación en complemento a 2

En el esquema de complemento a 2, un número negativo, K, se obtiene mediante la sustracción de su equivalente número positivo, P, de 2^n, de modo que $K = 2^n - P$. Si usamos nuestro ejemplo de cuatro bits, $-5 = 10000 - 0101 = 1011$ y $-3 = 10000 - 0011 = 1101$. Encontrar el complemento a 2 de esta forma requiere una sustracción que implica "pedir prestado". Sin embargo, puede observarse que si K_1 es el complemento a 1 de P y K_2 es el complemento a 2 de P, entonces

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

Se sigue que $K_2 = K_1 + 1$. Por tanto, una forma más simple de encontrar el complemento a 2 de un número consiste en sumar 1 a su complemento a 1, porque encontrar un complemento a 1 es sencillo. Es así como se obtienen los números en complemento a 2 en los circuitos lógicos que realizan operaciones aritméticas.

El lector habrá de desarrollar la habilidad para encontrar rápidamente números en complemento a 2. Hay una regla simple que sirve para tal propósito.

Regla para encontrar complementos a 2 Dado un número con signo, $B = b_{n-1}b_{n-2}, \ldots b_1b_0$, su complemento a 2, $K = k_{n-1}k_{n-2}\cdots k_1k_0$, se encuentra examinando los bits de B de derecha a izquierda y tomando la acción siguiente: se copian todos los bits de B que sean 0 y el primer bit que sea 1; luego simplemente se complementa el resto de los bits.

Por ejemplo, si $B = 0110$, entonces se copia $k_0 = b_0 = 0$ y $k_1 = b_1 = 1$ y se complementa el resto de modo que $k_2 = \overline{b}_2 = 0$ y $k_3 = \overline{b}_3 = 1$. Por tanto, $K = 1010$. Como otro ejemplo, si $B = 10110100$, entonces $K = 01001100$. La comprobación de esta regla se deja como ejercicio para el lector.

En la tabla 5.2 se ilustra la interpretación de los 16 patrones de cuatro bits en las tres representaciones de números con signo que hemos considerado. Nótese que tanto para la representación signo y magnitud como para la representación en complemento a 1 existen dos patrones que representan el valor cero. Para el complemento a 2 hay sólo uno de tales patrones. Además, obsérvese que el intervalo de los números que pueden representarse con cuatro bits en forma de complemento a 2 va de -8 a $+7$, mientras que en las otras dos representaciones va de -7 a $+7$.

Si se utiliza la representación en complemento a 2, un número $B = b_{n-1}b_{n-2}\cdots b_1b_0$ de n bits representa el valor

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad \textbf{[5.2]}$$

Por ende, el número negativo más grande, 100 . . . 00, tiene el valor -2^{n-1}. El número positivo más grande, 011 . . . 11, tiene el valor $2^{n-1} - 1$.

Tabla 5.2 Interpretación de enteros con signo de cuatro bits.

$b_3b_2b_1b_0$	Signo y magnitud	Complemento a 1	Complemento a 2
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

5.3.2 SUMA Y RESTA

Para valorar lo adecuado de las diferentes representaciones numéricas es preciso investigar sus usos en las operaciones, en particular la suma y la resta. Es posible ilustrar los aspectos buenos y malos de cada representación si consideramos números muy pequeños. Emplearemos números de cuatro bits, que constan del bit del signo y tres bits significativos. Por tanto, los números han de ser lo suficientemente pequeños para que la magnitud de su suma pueda expresarse en tres bits, lo que significa que la suma no puede superar el valor 7.

La suma de números positivos es igual en las tres representaciones numéricas. De hecho, es igual que la suma de números sin signo expuesta en la sección 5.2. Pero hay diferencias significativas cuando se trata de los números negativos. Las dificultades que surgen se evidencian si se consideran operandos con diferentes combinaciones de signos.

Suma en signo y magnitud

Si ambos operandos tienen el mismo signo, entonces la suma de los números con signo y magnitud es simple. Se suman las magnitudes y se da a la suma resultante el signo de los operandos. Sin embargo, la tarea se vuelve más complicada si los operandos tienen signos opuestos. Entonces es preciso sustraer el número más pequeño del más grande. Esto significa que también se necesitan circuitos lógicos que comparen y resten números. Dentro de poco veremos que es posible realizar restas sin la necesidad de estos circuitos. Por ello, la representación de signo y magnitud no se usa en las computadoras.

Suma en complemento a 1

Una ventaja obvia de la representación en complemento a 1 es que un número negativo se genera mediante la simple complementación de todos los bits del correspondiente número positivo. En la figura 5.9 se muestra lo que ocurre cuando se suman dos números. Hay que considerar cuatro casos en términos de diferentes combinaciones de signos. Como se observa en la mitad superior de la figura, el cálculo de $5 + 2 = 7$ y de $(-5) + 2 = (-3)$ es directo; una simple suma de los operandos produce el resultado correcto. Pero éste no es el caso con las otras dos posibilidades. El cálculo de $5 + (-2) = 3$ produce el vector bit 10010. Puesto que estamos usando números de cuatro bits, hay un acarreo de la posición del bit del signo. Además, los cuatro bits del resultado representan el número 2 en vez de 3, lo que es un resultado erróneo. Curiosamente, si se toma el acarreo de la posición del bit del signo y se suma al resultado en la posición del bit menos significativo, el nuevo resultado es la suma correcta de 3. Esta corrección se indica en gris oscuro en la figura. Una situación similar se presenta cuando se suman $(-5) + (-2) = (-7)$. Después de la suma inicial, el resultado es equivocado porque los cuatro bits de la suma son 0111, que representa $+7$ en vez de -7. Pero, de nuevo, existe un acarreo de la posición del bit del signo, que puede usarse para corregir el resultado sumándolo en la posición LSB, como se muestra en la figura 5.9.

La conclusión a partir de estos ejemplos es que la suma de números en complemento a 1 puede o no ser simple. En ciertos casos se requiere una corrección, lo que significa que debe realizarse una suma adicional. En consecuencia, el tiempo necesario para sumar dos números en complemento a 1 puede ser el doble del requerido para sumar dos números sin signo.

Suma en complemento a 2

Considérense las mismas combinaciones de números usados en el ejemplo de complemento a 1. En la figura 5.10 se indica cómo se realiza la suma con números en complemento a 2. La suma de $5 + 2 = 7$ y de $(-5) + 2 = (-3)$ es directa. El cálculo de $5 + (-2) = 3$ genera los cuatro bits correctos del resultado, 0011. Hay un acarreo de la posición del bit del signo, que simplemente puede ignorarse. El cuarto caso es $(-5) + (-2) = (-7)$. De nuevo, los cuatro bits del resultado, 1001, proporcionan la suma correcta (-7). En este caso también puede ignorarse el acarreo de la posición del bit del signo.

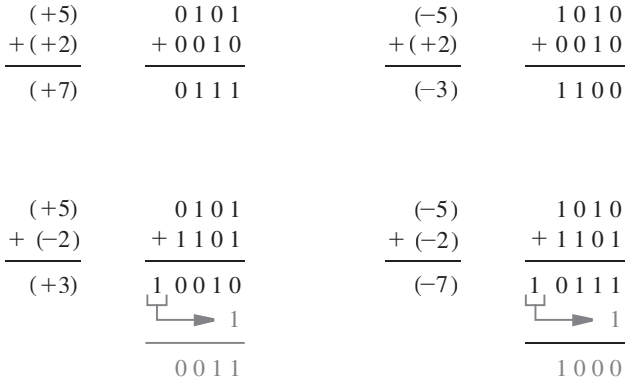


Figura 5.9 Ejemplos de suma en complemento a 1.

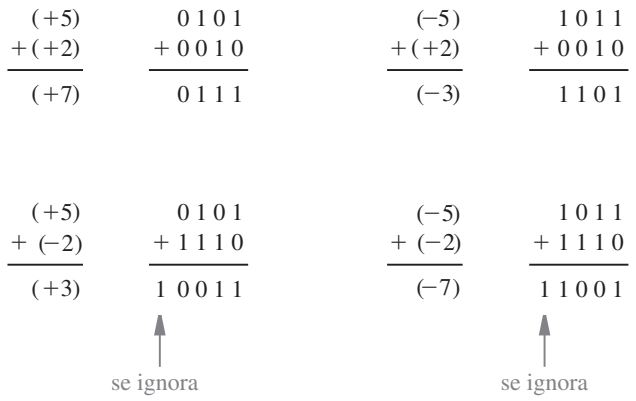


Figura 5.10 Ejemplos de suma en complemento a 2.

Como se ilustra con estos ejemplos, la suma de números en complemento a 2 es muy simple. Cuando los números se suman, el resultado siempre es correcto. Si hay un acarreo de la posición del bit del signo, simplemente se ignora. En consecuencia, el proceso de suma es el mismo, independientemente del signo de los operandos. Se puede realizar mediante un circuito sumador como el presentado en la figura 5.6. Por tanto, la notación en complemento a 2 es muy recomendable para la implementación de operaciones de suma. Ahora consideraremos su uso en operaciones de sustracción.

Resta en complemento a 2

La forma más sencilla de realizar restas es negar el sustraendo y sumarlo al minuendo. Esto se hace encontrando el complemento a 2 del sustraendo y luego realizando la suma. En la figura 5.11 se ilustra el proceso. La operación $5 - (+2) = 3$ supone encontrar el complemento a 2 de $+2$, que es 1110. Cuando se suma este número a 0101 el resultado es $0011 = (+3)$ y ocurre un acarreo de la posición del bit del signo, que se ignora. Una situación similar surge para $(-5) - (+2) = (-7)$. En los dos casos restantes no hay acarreo y el resultado es correcto.

Como un auxiliar gráfico para ver los ejemplos de suma y resta, en las figuras 5.10 y 5.11 colocamos todos los posibles patrones de cuatro bits en un círculo de módulo 16 dado en la figura 5.12. Si esos patrones de bit representasen enteros sin signo, serían los números de 0 a 15. Si representasen enteros en complemento a 2, entonces variarían de -8 a $+7$, como se muestra. La operación suma se realiza avanzando en el sentido de las manecillas del reloj para la magnitud del número que se va a sumar. Por ejemplo, $-5 + 2$ se determina comenzando en 1011 $(= -5)$ y dando dos pasos en el sentido de las manecillas del reloj, lo que produce el resultado 1101 $(= -3)$. La resta se realiza avanzando en contrasentido a las manecillas del reloj. Por ejemplo, $-5 - (+2)$ se determina empezando en 1011 y moviéndose dos pasos contra las manecillas del reloj, lo que produce 1001 $(= -7)$.

La conclusión clave de esta sección es que la operación resta puede realizarse como la operación suma, usando el complemento a 2 del sustraendo, sin importar los signos de los dos

$$\begin{array}{r} (+5) \\ -\ (+2) \\ \hline (+3) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ -\ 0\,0\,1\,0 \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0\,1\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,0\,1\,1 \end{array}$$

se ignora

$$\begin{array}{r} (-5) \\ -\ (+2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1\,0\,1\,1 \\ -\ 0\,0\,1\,0 \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1\,0\,1\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,1\,0\,0\,1 \end{array}$$

se ignora

$$\begin{array}{r} (+5) \\ -\ (-2) \\ \hline (+7) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ -\ 1\,1\,1\,0 \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 0\,1\,0\,1 \\ +\ 0\,0\,1\,0 \\ \hline 0\,1\,1\,1 \end{array}$$

$$\begin{array}{r} (-5) \\ -\ (-2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1\,0\,1\,1 \\ -\ 1\,1\,1\,0 \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{r} 1\,0\,1\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

Figura 5.11 Ejemplos de resta en complemento a 2.

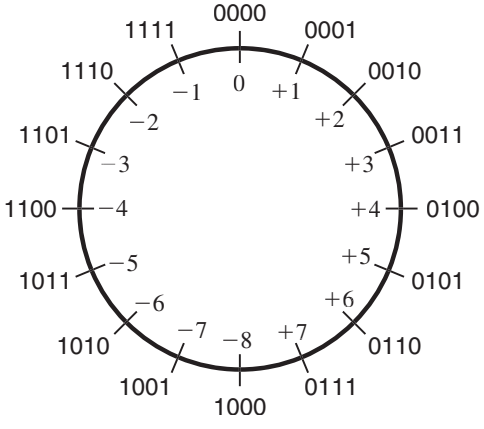


Figura 5.12 Interpretación gráfica de números de cuatro bits en complemento a 2.

operandos. Por tanto, debe ser posible utilizar el mismo circuito sumador para efectuar tanto suma como resta.

5.3.3 UNIDAD SUMADORA Y RESTADORA

La única diferencia entre realizar sumas y restas es que para la resta es necesario usar el complemento a 2 de un operando. Sean X y Y los dos operandos, tal que Y funciona como el sustraendo en la resta. En la sección 5.3.1 aprendimos que el complemento a 2 se obtiene sumando 1 al complemento a 1 de Y. Sumar 1 en la posición del bit menos significativo puede lograrse simplemente poniendo el bit de acarreo c_0 en 1. El complemento a 1 de un número se obtiene complementando cada uno de sus bits. Esto podría hacerse con compuertas NOT, pero necesitamos un circuito más flexible donde pueda emplearse el valor verdadero de Y para la suma y su complemento para la resta.

En la sección 5.2 explicamos que es posible usar compuertas XOR de dos entradas para elegir entre versiones verdadera y complementada de un valor de entrada, bajo el control de la otra entrada. Esta idea puede aplicarse en el diseño de la unidad sumadora/restadora como sigue. Supóngase que existe una señal de control que elige si ha de realizarse suma o resta. Llámese esta señal $\overline{\text{Add}}$/Sub. Además, sea 0 su valor para la suma y 1 para la resta. Para indicar este hecho se coloca una barra sobre Add. Ésta es una convención usada de manera común, donde una barra sobre un nombre significa que la acción especificada por el nombre se toma si la señal de control tiene el valor 0. Ahora, conéctese cada bit de Y a una entrada de una compuerta XOR, con la otra entrada conectada a $\overline{\text{Add}}$/Sub. Las salidas de las compuertas XOR representan Y si $\overline{\text{Add}}/\text{Sub} = 0$, y representan el complemento a 1 de Y si $\overline{\text{Add}}/\text{Sub} = 1$. Esto conduce al circuito de la figura 5.13, cuya parte principal es un sumador de n bits, el cual puede implementarse con la estructura de acarreo en cascada de la figura 5.6. Nótese que la señal de control $\overline{\text{Add}}$/Sub también se conecta al

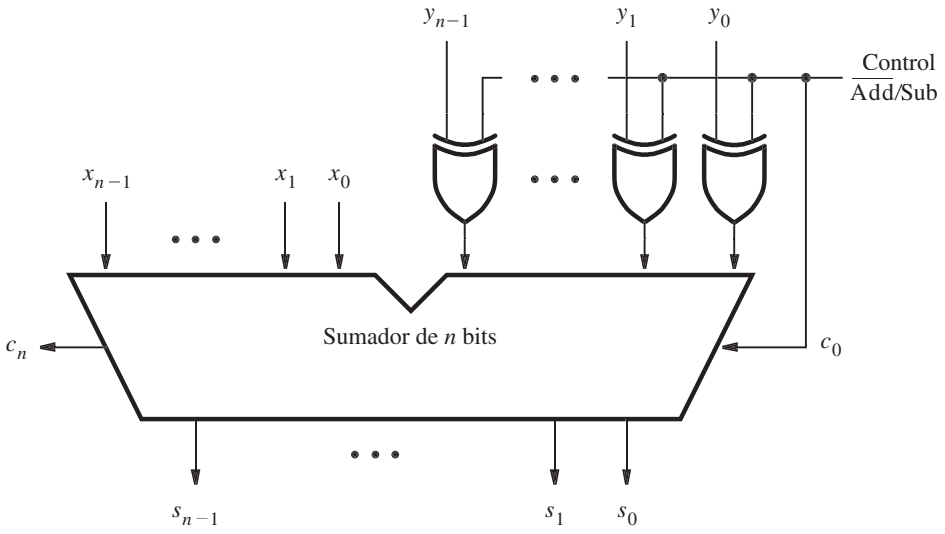


Figura 5.13 Unidad sumadora/restadora.

acarreo c_0. Esto hace $c_0 = 1$ cuando se realiza la resta; por tanto, suma el 1 que se necesita para formar el complemento a 2 de *Y*. Cuando se haga la operación suma se tendrá $c_i = 0$.

La unidad combinada sumador/restador es un buen ejemplo de un concepto importante en el diseño de circuitos lógicos. Es útil diseñar circuitos para que sean lo más flexibles posible y para explotar las partes comunes de los circuitos para cuantas tareas sea factible. Este enfoque minimiza el número de compuertas necesarias para implementar tales circuitos, al tiempo que reduce sustancialmente la complejidad del cableado.

5.3.4 ESQUEMA DE COMPLEMENTO A LA BASE (RAÍZ)

La idea de realizar una resta mediante la suma de un complemento del sustraendo no se restringe a los números binarios. El esquema del complemento a 2 se entiende mejor si consideramos su contraparte en el sistema numérico decimal. Considérese la resta de números decimales de dos dígitos. Calcular un resultado como $74 - 33 = 41$ es simple porque cada dígito del sustraendo es más pequeño que el dígito correspondiente del minuendo; por tanto, no se necesita "pedir prestado" al restar el dígito menos significativo. Pero restar $74 - 36 = 38$ no es tan sencillo porque se requiere "pedir prestado" una vez al restar el dígito menos significativo. Si se "pide prestado", el cálculo se vuelve más complicado.

Supóngase que el cálculo requerido se reestructura del modo siguiente

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Ahora se necesitan dos restas. Restar 36 de 100 aún supone "pedir prestado" una vez. Pero si observamos que $100 = 99 + 1$, ello puede evitarse si escribimos

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

La resta entre paréntesis no requiere "pedir prestado"; se realiza restando de 9 cada dígito del sustraendo. Es posible observar una correlación directa entre esta expresión y la utilizada para el complemento a 2, como se refleja en el circuito de la figura 5.13. La operación (99 – 36) es análoga a complementar el sustraendo *Y* para encontrar su complemento a 1, que es lo mismo que restar cada bit de 1. Si utilizamos números decimales se encuentra el *complemento* a 9 del sustraendo restando cada dígito de 9. En la figura 5.13 se suma el acarreo de 1 para formar el complemento a 2 de *Y*. En nuestro ejemplo decimal se realiza $(99 - 36) + 1 = 64$. Aquí 64 es el complemento a 10 de 36. Para un número decimal de *n* dígitos, *N*, su *complemento a 10,* K_{10}, se define como $K_{10} = 10^n - N$, mientras que su complemento a 9, K_9, es $K_9 = (10^n - 1) - N$.

Por ende, la resta requerida $(74 - 36)$ puede realizarse al sumar el complemento a 10 del sustraendo, como en

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

La resta 138 − 100 es trivial porque significa que el primer dígito de 138 simplemente se borra. Esto es análogo a ignorar el acarreo del circuito en la figura 5.13, como explicamos para los ejemplos de resta de la figura 5.11.

Ejemplo 5.1 Suponga que A y B son números decimales de n dígitos. Si se usa el anterior enfoque de complemento a 10, B puede restarse de A del modo siguiente:

$$A - B = A + (10^n - B) - 10^n$$

Si $A \geq B$, entonces la operación $A + (10^n - \text{B})$ produce un acarreo de 1, el cual es equivalente a 10^n; por tanto, simplemente se puede ignorar.

Pero si A < B, entonces la operación A + $(10^n - \text{B})$ produce un acarreo de 0. Sea M el resultado obtenido, de modo que

$$A - B = M - 10^n$$

Esto puede volverse a escribir como

$$10^n - (B - A) = M$$

El miembro izquierdo de esta ecuación es el complemento a 10 de $(B - A)$. El complemento a 10 de un número positivo representa un número negativo que tiene la misma magnitud. Por ende, M representa correctamente el valor negativo obtenido del cálculo $A - B$ cuando $A < B$. Este concepto se ilustra en los ejemplos siguientes.

Ejemplo 5.2 Cuando se usan números binarios con signo se emplea 0 en el bit que está más a la izquierda para denotar un número positivo y 1 para indicar un número negativo. Si se quiere construir hardware que opere con números decimales con signo, podría usarse un enfoque similar. Sea 0 en la posición del dígito más a la izquierda un número positivo y 9 uno negativo. Note que 9 es el complemento a 9 de 0 en el sistema decimal, así como 1 es el complemento a 1 de 0 en el sistema binario.

Por tanto, al usar números de tres dígitos con signo, $A = 045$ y $B = 027$ son números positivos con magnitudes 45 y 27, respectivamente. El número B puede restarse de A del modo siguiente

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999 - 027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

Esto proporciona la respuesta correcta: +18.

Considere a continuación el caso donde el minuendo tiene un valor menor que el sustraendo, lo cual se ilustra mediante el cálculo

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

A partir de esta expresión parece que tendremos que realizar la resta 982 − 1000. Pero como vimos en el ejemplo 5.1, esto puede volverse a escribir como

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

En consecuencia, 982 es el número negativo que resulta cuando se forma el complemento a 10 de $(A - B)$. Con base en el cálculo previo se sabe que $(A - B) = 018$, que indica $+18$. Por tanto, el número con signo 982 es la representación en complemento a 10 de -18, que es el resultado requerido.

Ejemplo 5.3

Sean $C = 955$ y $D = 973$; por tanto, los valores de C y D son -45 y -27, respectivamente. El número D puede restarse de C del modo siguiente

$$\begin{aligned} C - D &= 955 - 973 \\ &= 955 + 1000 - 1000 - 973 \\ &= 955 + (999 - 973) + 1 - 1000 \\ &= 955 + 026 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

El número 982 es la representación en complemento a 10 de -18, que es el resultado correcto.

Considere ahora el caso $D - A$, donde $D = 973$ y $A = 045$:

$$\begin{aligned} D - A &= 973 - 045 \\ &= 973 + 1000 - 1000 - 045 \\ &= 973 + (999 - 045) + 1 - 1000 \\ &= 973 + 954 + 1 - 1000 \\ &= 1928 - 1000 \\ &= 928 \end{aligned}$$

El resultado 928 es la representación en complemento a 10 de -72.

Estos ejemplos ilustran que los números con signo pueden restarse sin una operación de resta que implique "pedir prestado". La única resta que se necesita es para formar el complemento a 9 del sustraendo, caso en el que cada dígito simplemente se resta de 9. Por ende, un circuito

que forme el complemento a 9, combinado con un sumador normal, será suficiente tanto para la suma como para la resta de números decimales con signo. Un punto clave es que el hardware necesita trabajar sólo con n dígitos si se usan números de n dígitos. Cualquier acarreo que pueda generarse desde la posición del dígito más a la izquierda simplemente se ignora.

El concepto de restar un número mediante la suma de su complemento a la base es general. Si la base es r, entonces el complemento a r, K_r, de un número de n dígitos, N, se determina como $K_r = r^n - N$. El complemento a $(r - 1)$, K_{r-1}, se define como $K_{r-1} = (r^n - 1) - N$; se calcula simplemente restando cada dígito de N del valor $(r - 1)$. El complemento a $(r - 1)$ se denomina complemento a la base disminuida. Los circuitos para formar los complementos a $(r - 1)$ son más simples que los de resta general que implican "pedir prestado". Los circuitos son particularmente simples en el caso binario, donde el complemento a 1 requiere sólo la inversión de cada bit.

Ejemplo 5.4 En la figura 5.11 ilustramos la operación resta sobre números binarios dados en representación de complemento a 2. Considere el cálculo $(+5) - (+2) = (+3)$ con el enfoque expuesto antes. Cada número está representado por un patrón de cuatro bits. El valor 2^4 se representa como 10000. Luego

$$\begin{aligned} 0101 - 0010 &= 0101 + (10000 - 0010) - 10000 \\ &= 0101 + (1111 - 0010) + 1 - 10000 \\ &= 0101 + 1101 + 1 - 10000 \\ &= 10011 - 10000 \\ &= 0011 \end{aligned}$$

Puesto que $5 > 2$, hay un acarreo de la posición del cuarto bit. Representa el valor 2^4, que se denota mediante el patrón 10000.

Ejemplo 5.5 Considere ahora el cálculo $(+2) - (+5) = (-3)$, que produce

$$\begin{aligned} 0010 - 0101 &= 0010 + (10000 - 0101) - 10000 \\ &= 0010 + (1111 - 0101) + 1 - 10000 \\ &= 0010 + 1010 + 1 - 10000 \\ &= 1101 - 10000 \end{aligned}$$

Puesto que $2 < 5$, no hay acarreo de la posición del cuarto bit. La respuesta, 1101, es la representación en complemento a 2 de -3. Note que

$$\begin{aligned} 1101 &= 10000 + 0010 - 0101 \\ &= 10000 - (0101 - 0010) \\ &= 10000 - 0011 \end{aligned}$$

que indica que 1101 es el complemento a 2 de 0011 $(+3)$.

Ejemplo 5.6

Por último, considere el caso donde el sustraendo es un número negativo. El cálculo $(+5) - (-2) = (+7)$ se realiza del modo siguiente:

$$\begin{aligned} 0101 - 1110 &= 0101 + (10000 - 1110) - 10000 \\ &= 0101 + (1111 - 1110) + 1 - 10000 \\ &= 0101 + 0001 + 1 - 10000 \\ &= 0111 - 10000 \end{aligned}$$

Aunque $5 > (-2)$, el patrón 1110 es mayor que el patrón 0101 cuando los patrones se tratan como números sin signo. Por tanto, no hay acarreo de la posición del cuarto bit. La repuesta 0111 es la representación en complemento a 2 de $+7$. Note que

$$\begin{aligned} 0111 &= 10000 + 0101 - 1110 \\ &= 10000 - (1110 - 0101) \\ &= 10000 - 1001 \end{aligned}$$

y 1001 representa -7.

5.3.5 DESBORDAMIENTO ARITMÉTICO

Se supone que el resultado de la suma o la resta encaja dentro de los bits significativos utilizados para representar los números. Si se usan n bits para representar números con signo, entonces el resultado debe estar en el intervalo que va de -2^{n-1} a $2^{n-1} - 1$. Si el resultado no se halla en ese intervalo, entonces se dice que ocurrió un *desbordamiento aritmético*. Para garantizar la operación correcta de un circuito aritmético es importante detectar cuándo ocurre el desbordamiento.

En la figura 5.14 se presentan los cuatro casos donde se suman números en complemento a 2 con magnitudes de 7 y 2. Puesto que estamos empleando números de cuatro bits, hay tres bits significativos, b_{2-0}. Cuando los números tienen signos opuestos, no hay desbordamiento. Pero si

(+7)	0 1 1 1	(−7)	1 0 0 1
+(+2)	+ 0 0 1 0	+(+2)	+ 0 0 1 0
(+9)	1 0 0 1	(−5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(−7)	1 0 0 1
+ (−2)	+ 1 1 1 0	+ (−2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(−9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figura 5.14 Ejemplos de determinación de desbordamiento.

ambos números tienen el mismo signo, la magnitud del resultado es 9, que no puede representarse sólo con tres bits significativos; por tanto, hay desbordamiento. La clave para determinar si éste se presenta es el acarreo proveniente de la posición del MSB, llamado c_3 en la figura, y el proveniente de la posición del bit del signo, llamado c_4. En la figura se indica que hay desbordamiento cuando esos acarreos tienen diferentes valores, y se produce una suma correcta cuando tienen el mismo valor. De hecho, esto es cierto en general tanto para la suma como para la resta de números en complemento a 2. Como comprobación rápida de esta afirmación, considérense los ejemplos de la figura 5.10, donde los números son lo suficientemente pequeños como para que no haya desbordamiento en ningún caso. En los dos ejemplos de la parte superior de la figura, existe un acarreo de 0 proveniente de las posiciones del signo y del MSB. En los dos ejemplos inferiores, hay un acarreo de 1 proveniente de ambas posiciones. En consecuencia, para los ejemplos de las figuras 5.10 y 5.14 el desbordamiento se detecta mediante

$$\begin{aligned}\text{Desbordamiento} &= c_3\overline{c}_4 + \overline{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

Para números de n bits se tiene

$$\text{Desbordamiento} = c_{n-1} \oplus c_n$$

Por ende, el circuito de la figura 5.13 puede modificarse para incluir comprobación de desbordamiento con la adición de una compuerta XOR.

5.3.6 PROBLEMAS DE RENDIMIENTO

Cuando se compra un sistema digital, como una computadora, el comprador presta atención especial al rendimiento que espera le proporcione el sistema y al costo de adquirirlo. El rendimiento superior casi siempre implica mayor costo. Sin embargo, un gran aumento en el rendimiento puede lograrse con un modesto incremento en el costo. Un indicador del valor de un sistema usado comúnmente es su *razón precio/rendimiento*.

La suma y la resta de números son operaciones fundamentales que se realizan con frecuencia en el curso de un cálculo. La velocidad con la que se evalúan tiene un efecto enorme en el rendimiento total de una computadora. A la luz de esto, echaremos un vistazo cercano a la velocidad de la unidad sumadora/restadora de la figura 5.13. Nos interesa el retraso más largo desde el momento en que los operandos X y Y se presentan como entradas, hasta el instante en que todos los bits de la suma S y el acarreo final, c_n, son válidos. La mayor parte de este retraso lo provoca el circuito sumador de n bits. Supóngase que el sumador se implementa con la estructura de acarreo en cascada de la figura 5.6 y que cada etapa del sumador completo es el circuito de la figura 5.4*c*. El retraso para la señal de acarreo en este circuito, Δt, es igual a dos retrasos de compuerta. En la sección 5.2.2 indicamos que el resultado final de la suma será válido luego de un retraso de $n\Delta t$, que es igual a $2n$ retrasos de compuerta. Además del retraso en la trayectoria del acarreo en cascada, también hay un retraso en las compuertas XOR que alimentan el valor verdadero o el complementado de Y a las entradas del sumador. Si este retraso es igual a un retraso de compuerta, entonces el retraso total del circuito en la figura 5.13 es $2n + 1$ retrasos de compuerta. Para un n grande, digamos $n = 32$ o $n = 64$, el retraso conduciría a un rendimiento inaceptablemente malo. Por tanto, es importante encontrar los circuitos más veloces para realizar la suma.

La velocidad de cualquier circuito está limitada por el retraso mayor a lo largo de las trayectorias a través del circuito. En el caso del circuito de la figura 5.13, el retraso más grande está a

lo largo de la trayectoria que va de la entrada y_i, pasa por la compuerta XOR y por el circuito de acarreo de cada etapa del sumador. El retraso más largo se conoce como *retraso de trayectoria crítica*, y la trayectoria que lo ocasiona se llama *trayectoria crítica*.

5.4 SUMADORES VELOCES

El rendimiento de un gran sistema digital depende de la velocidad de los circuitos que forman sus diversas unidades funcionales. Es obvio que el mejor rendimiento se logra con circuitos más rápidos, los cuales se consiguen empleando tecnología superior (casi siempre más nueva) en la que los retrasos en las compuertas básicas se reducen. Pero también puede lograrse modificando la estructura global de una unidad funcional, lo cual puede conducir incluso a un rendimiento más impresionante. En esta sección explicaremos una posibilidad para implementar un sumador de *n* bits que reduce sustancialmente el tiempo necesario para sumar números.

5.4.1 SUMADOR CON ACARREO DE ADELANTO

Para reducir el retraso producido por el efecto de propagación de acarreo a través del sumador con acarreo en cascada puede evaluarse rápidamente si en cada etapa el acarreo proveniente de la etapa previa tendrá valor 0 o 1. Si es posible hacer una evaluación correcta en un tiempo hasta cierto punto breve, entonces se mejorará el rendimiento de todo el sumador.

A partir de la figura 5.4*b*, la función acarreo para la etapa *i* puede realizarse como

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Si esta expresión se factoriza como

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

entonces se puede escribir como

$$c_{i+1} = g_i + p_i c_i \qquad \textbf{[5.3]}$$

donde

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

La función g_i es igual a 1 cuando ambas entradas, x_i y y_i, son iguales a 1, independientemente del valor del acarreo entrante a esta etapa, c_i. Como en este caso está garantizado que la etapa *i* generará un acarreo, a *g* se le llama función *generada*. La función p_i es igual a 1 cuando al menos una de las entradas, x_i y y_i, es igual a 1. En este caso se produce un acarreo si $c_i = 1$. El efecto es que el acarreo de 1 se propaga a lo largo de la etapa *i*; por tanto, p_i se denomina función *propagada*.

Al expandir la expresión 5.3 en términos de la etapa $i - 1$ se produce

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

La misma expansión para las otras etapas, finalizando con la etapa 0, produce

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \qquad \textbf{[5.4]}$$

Esta expresión representa un circuito AND-OR de dos niveles en el que c_{i+1} se evalúa muy rápidamente. Un sumador basado en esta expresión se denomina *sumador con acarreo de adelanto*.

Para apreciar el significado físico de la expresión 5.4 es necesario considerar su efecto en la construcción de un sumador veloz en comparación con los detalles del sumador con acarreo en cascada. Así lo haremos al examinar la estructura detallada de las dos etapas que suman los bits menos significativos: las etapas 0 y 1. En la figura 5.15 se muestran las primeras dos etapas de un sumador con acarreo en cascada en el que las funciones de acarreo se implementan como se indica en la expresión 5.3. Cada etapa es en esencia el circuito de la figura 5.4*c*, excepto que

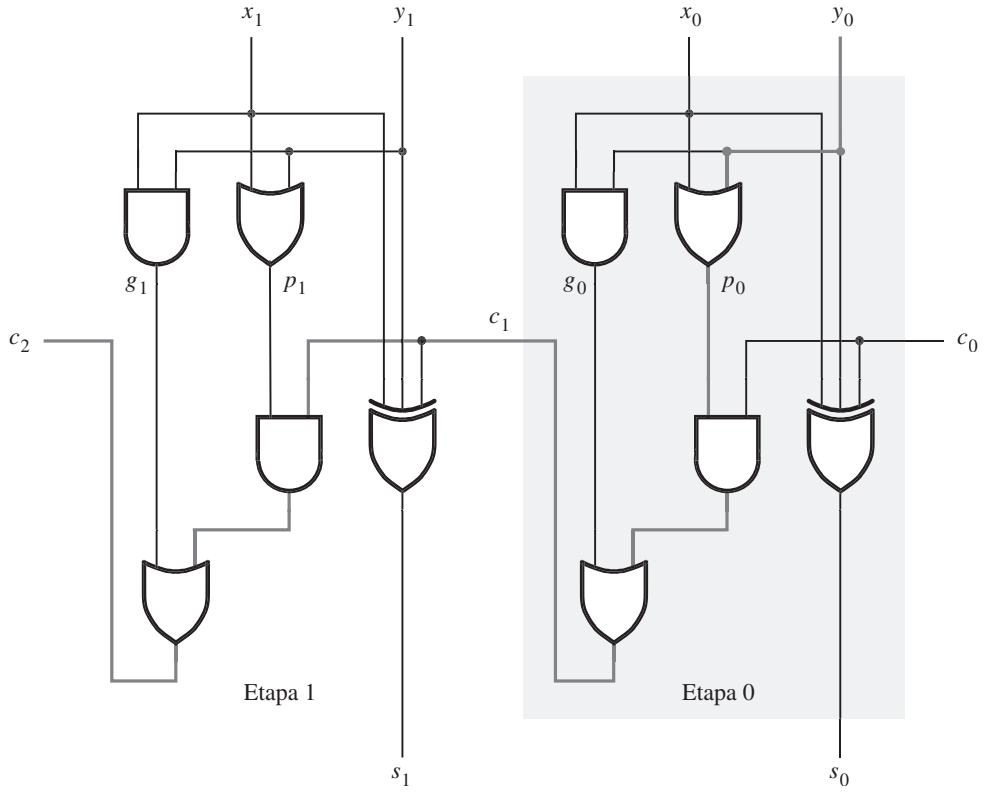


Figura 5.15 Sumador con acarreo en cascada basado en la expresión 5.3.

se usa una compuerta OR adicional (que produce la señal p_i), en lugar de una compuerta AND porque se factorizó la expresión de suma de productos para c_{i+1}.

La lentitud del sumador con acarreo en cascada es producto de la larga trayectoria por la que una señal de acarreo debe propagarse. En la figura 5.15 la trayectoria crítica va de las entradas x_0 y y_0 a la salida c_2. Pasa por cinco compuertas, resaltadas en gris oscuro. La trayectoria en las otras etapas de un sumador de n bits es la misma que en la etapa 1. Por tanto, el retraso total a lo largo de la trayectoria crítica es $2n + 1$.

En la figura 5.16 se presentan las primeras dos etapas del sumador con acarreo de adelanto; se usa la expresión 5.4 para implementar las funciones de acarreo. En consecuencia

$$c_1 = g_0 + p_0c_0$$
$$c_2 = g_1 + p_1g_0 + p_1p_0c_0$$

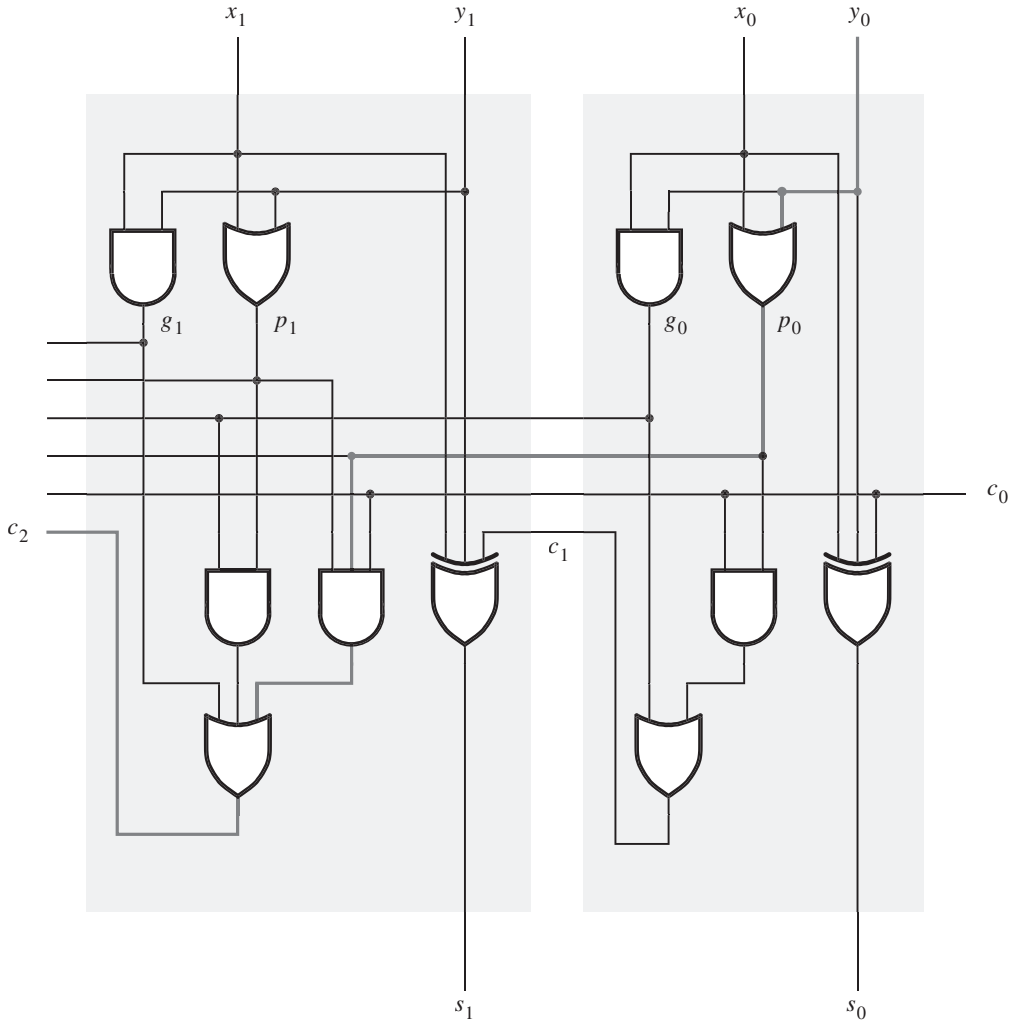


Figura 5.16 Primeras dos etapas de un sumador con acarreo de adelanto.

La trayectoria crítica para producir la señal c_2 se resalta en gris oscuro. En este circuito, c_2 se produce tan rápido como c_1, después de un total de tres retrasos de compuerta. Si extendemos el circuito a *n* bits, la señal de acarreo final c_n también se producirá después de sólo tres retrasos de compuerta, ya que la expresión 5.4 es un gran circuito de dos niveles (AND-OR).

El retraso total en el sumador de *n* bits con acarreo de adelanto es de cuatro retrasos de compuerta. Los valores de todas las señales g_i y p_i se determinan después de un retraso de compuerta. Se precisan dos retrasos de compuerta más para evaluar todas las señales de acarreo. Finalmente, se requiere un retraso de compuerta más (XOR) para generar todos los bits suma. La clave para el buen rendimiento del sumador es la evaluación rápida de las señales de acarreo.

La complejidad de un sumador de *n* bits con acarreo de adelanto aumenta rápidamente conforme *n* se vuelve más grande. Para reducir la complejidad es posible aplicar un enfoque *jerárquico* al diseñar sumadores grandes. Supóngase que queremos diseñar un sumador de 32 bits. Este sumador puede dividirse en cuatro bloques de ocho bits, de tal modo que los bits $b_{7\text{-}0}$ sean el bloque 0, los bits $b_{15\text{-}8}$ sean el bloque 1, los bits $b_{23\text{-}16}$ sean el bloque 2 y los bits $b_{31\text{-}24}$ sean el bloque 3. Luego podemos implementar cada bloque como un sumador de ocho bits con acarreo de adelanto. Las señales de acarreo provenientes de los cuatro bloques son c_8, c_{16}, c_{24} y c_{32}. Ahora tenemos dos posibilidades. Se pueden conectar los cuatro bloques como cuatro etapas en un sumador con acarreo en cascada. Por ende, mientras el acarreo de adelanto se use dentro de cada bloque, el acarreo cae en cascada entre los bloques. Este circuito se ilustra en la figura 5.17.

En vez de usar un enfoque de acarreo en cascada entre bloques es posible diseñar un circuito más veloz en el que se realice un acarreo de adelanto de segundo nivel para producir rápidamente las señales de acarreo entre bloques. La estructura de este "sumador jerárquico con acarreo de adelanto" se muestra en la figura 5.18. Cada bloque de la fila superior incluye un sumador de ocho bits con acarreo de adelanto, basado en las señales generadas, g_i, y las señales propagadas, p_i, para cada etapa en el bloque, como explicamos líneas arriba. Sin embargo, en vez de producir una señal de acarreo proveniente del bit más significativo del bloque, cada bloque produce señales generada y propagada para todo el bloque. Sean G_j y P_j dichas señales para cada bloque *j*. Ahora G_j y P_j se usan como entradas a un circuito con acarreo de adelanto de segundo nivel —parte inferior de la figura 5.18— que evalúe todos los acarreos entre bloques. Es posible derivar las señales de bloque generada y propagada para el bloque 0 examinando la expresión para c_8

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

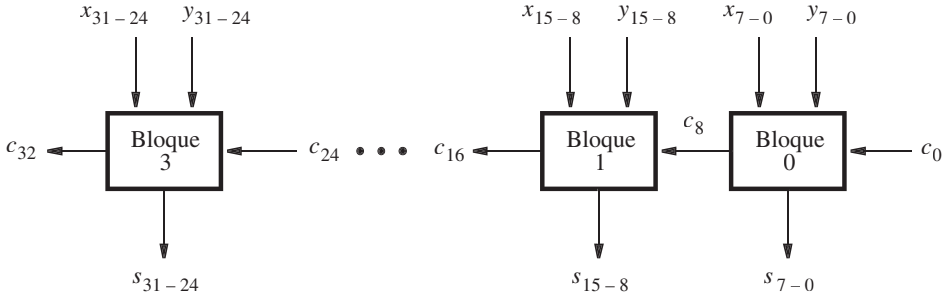


Figura 5.17 Sumador jerárquico con acarreo de adelanto con acarreo en cascada entre bloques.

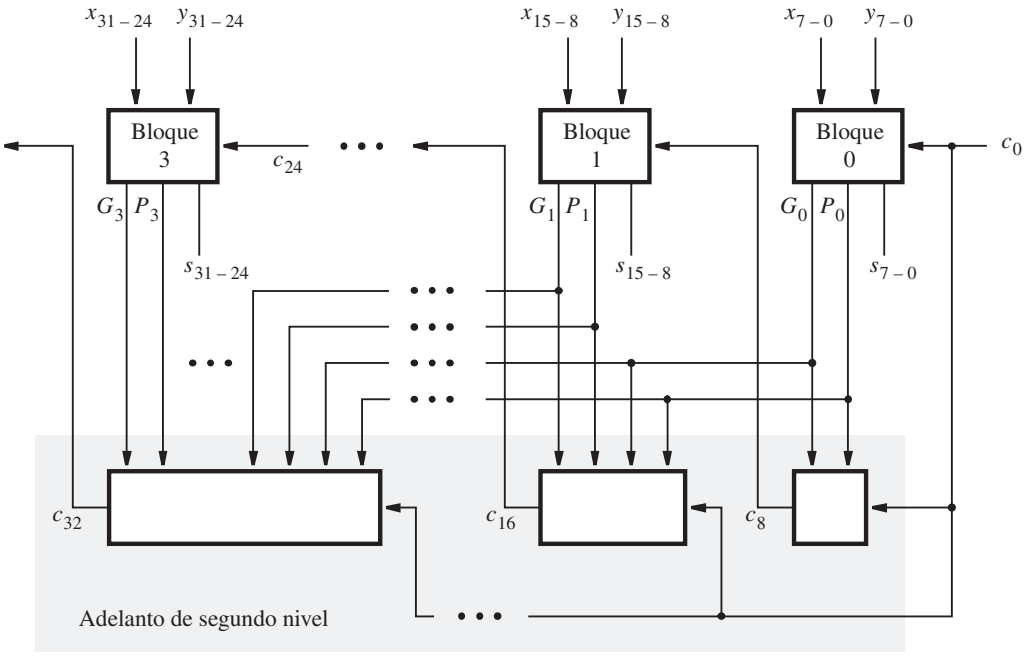


Figura 5.18 Sumador jerárquico con acarreo de adelanto.

El último término de esta expresión indica que si las ocho funciones propagadas son 1, entonces el acarreo c_0 se propaga por todo el bloque. En consecuencia

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

El resto de los términos de la expresión para c_8 representan todos los demás casos cuando el bloque produce un acarreo. Por tanto

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

La expresión para c_8 en el sumador jerárquico está dada por

$$c_8 = G_0 + P_0c_0$$

Para el bloque 1, las expresiones para G_1 y P_1 tienen la misma forma que para G_0 y P_0, excepto que cada subíndice i se sustituye con $i + 8$. Las expresiones para G_2, P_2, G_3 y P_3 se deducen de la misma forma. La expresión para el acarreo del bloque 1, c_{16}, es

$$\begin{aligned} c_{16} &= G_1 + P_1c_8 \\ &= G_1 + P_1G_0 + P_1P_0c_0 \end{aligned}$$

De manera similar, las expresiones para c_{24} y c_{32} son

$$\begin{aligned} c_{24} &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0 \\ c_{32} &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0 \end{aligned}$$

Si se utiliza este esquema se precisan dos retrasos de compuerta más para producir las señales de acarreo c_8, c_{16} y c_{24} que el tiempo necesario para generar las funciones G_j y P_j. Por consiguiente, como G_j y P_j requieren tres retrasos de compuerta, c_8, c_{16} y c_{24} quedan disponibles después de cinco retrasos de compuerta. El tiempo necesario para sumar dos números de 32 bits supone estos cinco retrasos de compuerta más dos adicionales para producir los acarreos internos en los bloques 1, 2 y 3, más un retraso de compuerta adicional (XOR) para generar todos los bits suma. Esto da un total de ocho retrasos de compuerta.

En la sección 5.3.5 establecimos que se requieren $2n + 1$ retrasos de compuerta para sumar dos números con el sumador con acarreo en cascada. Para números de 32 bits esto implica 65 retrasos de compuerta. Es evidente que el sumador con acarreo de adelanto ofrece una mejora mayor en el rendimiento. A cambio, el circuito requerido tiene una complejidad mucho mayor.

Consideraciones tecnológicas

El anterior análisis de retraso supone que es posible utilizar compuertas con cualquier número de entradas. En los capítulos 3 y 4 explicamos que la tecnología empleada para implementar las compuertas limita la carga de entrada a un número más bien pequeño de entradas. Por tanto, hay que tomar en cuenta la realidad de las restricciones en la carga de entrada. Para ilustrar este problema, considérense las expresiones para los primeros ocho acarreos:

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\
&\vdots \\
c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\
&\quad + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0
\end{aligned}$$

Supóngase que la máxima entrada de carga de las compuertas es de cuatro entradas. Entonces es imposible implementar todas estas expresiones con un circuito AND-OR de dos niveles. El mayor problema es c_8, donde una de las compuertas AND requiere nueve entradas; más aún, la compuerta OR también precisa nueve entradas. Para satisfacer la restricción de la carga de entrada la expresión para c_8 puede volverse a escribir como

$$\begin{aligned}
c_8 = {} & (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\
& + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0
\end{aligned}$$

Para implementar esta expresión se necesitan 11 compuertas AND y tres OR. El retraso de propagación para generar c_8 consta de un retraso de compuerta para desarrollar todas las g_i y p_i, dos retrasos de compuerta para producir los términos suma de productos entre paréntesis, un retraso de compuerta para formar el término producto entre corchetes y un retraso para la operación OR final de los términos. En consecuencia, c_8 es válida después de cinco retrasos de compuerta, en lugar de tres retrasos de compuerta que se necesitarían sin la restricción de carga de entrada.

Puesto que las limitaciones de carga de entrada reducen la velocidad del sumador con acarreo de adelanto, algunos dispositivos que se caracterizan por baja entrada de carga incluyen circuitos dedicados para la implementación de sumadores veloces. Ejemplos de tales dispositivos incluyen los FPGA, cuyos bloques lógicos se basan en tablas de consulta.

Antes de dejar el tema de los sumadores con acarreo de adelanto hay que considerar otra implementación de la estructura de la figura 5.16. Es posible obtener la misma funcionalidad con el circuito de la figura 5.19. En este caso, la etapa 0 se implementa con el circuito de la

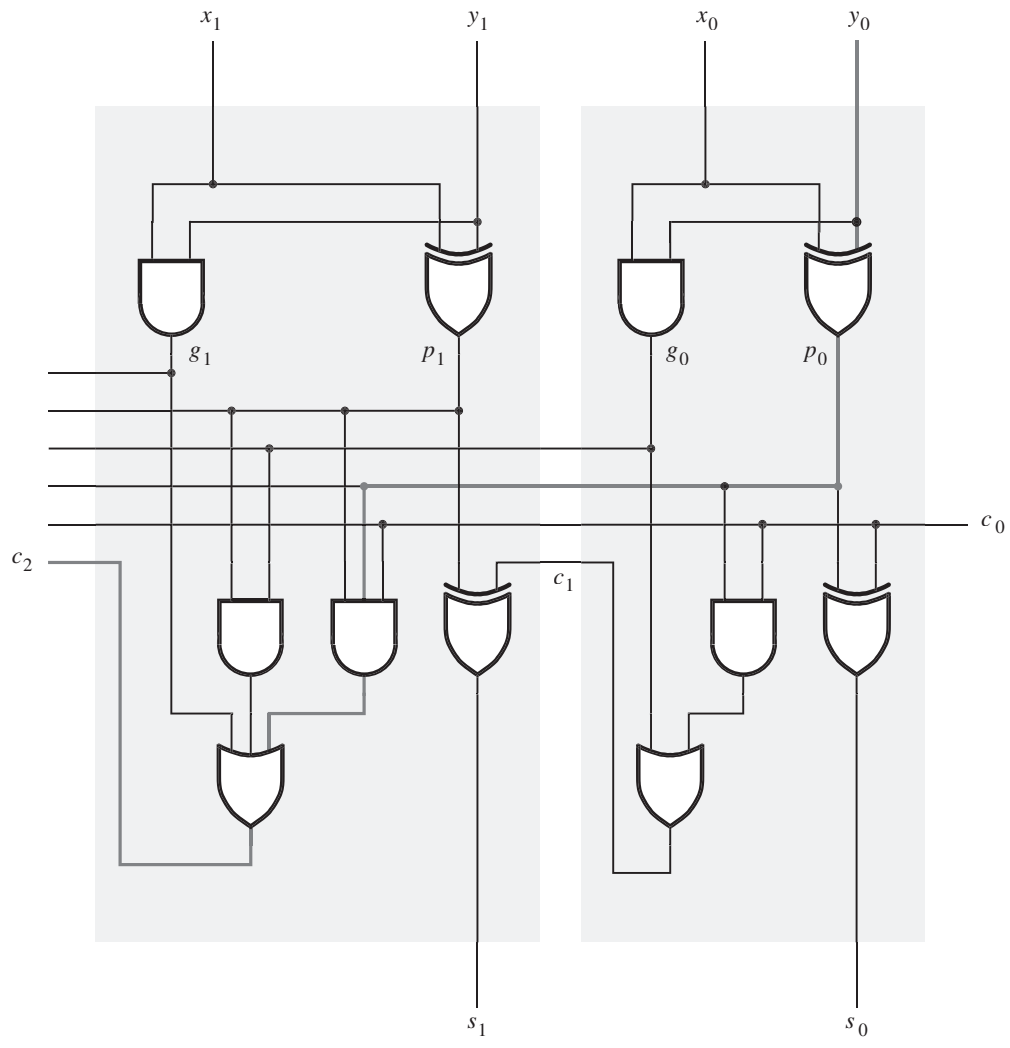


Figura 5.19 Otro diseño para un sumador con acarreo de adelanto.

figura 5.5, en el que se usan dos compuertas XOR de dos entradas para generar el bit suma, en vez de tener 1 compuerta XOR de tres entradas. La salida de la primera compuerta XOR también puede servir como la señal propagada p_0. Por tanto, no se necesita la correspondiente compuerta OR de la figura 5.16. La etapa 1 se construye siguiendo el mismo enfoque.

Los circuitos de las figuras 5.16 y 5.19 requieren el mismo número de compuertas. ¿Pero alguno de ellos es mejor en algún sentido? La respuesta debe buscarse al considerar los aspectos específicos de la tecnología usada para implementar los circuitos. Si se emplearon un CPLD o un FPGA, como los de las figuras 3.33 y 3.39, entonces no importa cuál circuito se elija. Con una macrocelda puede realizarse una función XOR de tres entradas en el CPLD, utilizando la

expresión en suma de productos

$$s_i = x_i\overline{y}_i\overline{c}_i + \overline{x}_i y_i \overline{c}_i + \overline{x}_i\overline{y}_i c_i + x_i y_i c_i$$

porque la macrocelda permite la implementación de cuatro términos producto.

En el FPGA, cualquier función de tres entradas puede implementarse en una sola celda lógica; por tanto, es fácil realizar una XOR de tres entradas. Sin embargo, supóngase que deseamos construir un sumador con acarreo de adelanto en un chip a la medida. Si la compuerta XOR se construye con el enfoque expuesto en la sección 3.9.1, entonces en realidad se implementaría una XOR de tres entradas usando dos compuertas XOR de dos entradas, como se hizo para los bits suma de la figura 5.19. Por ende, si la primera compuerta XOR realiza la función $x_i \oplus y_i$, que también es la función propagada p_i, entonces es obvio que la alternativa presentada en la figura 5.19 es más atractiva. El punto importante de esta explicación es que la optimización de los circuitos lógicos puede depender de la tecnología destino. Las herramientas CAD toman en cuenta este hecho.

El sumador con acarreo de adelanto es un concepto bien conocido. Existen chips estándar que implementan una parte del circuito de acarreo de adelanto. Se llaman *generadores de acarreo de adelanto*. Las herramientas CAD suelen incluir subcircuitos prediseñados para sumadores, que los diseñadores pueden usar para diseñar unidades más grandes.

5.5 DISEÑO DE CIRCUITOS ARITMÉTICOS CON EL USO DE HERRAMIENTAS CAD

En esta sección mostraremos cómo diseñar circuitos aritméticos con herramientas CAD. Analizaremos dos métodos de diseño: uso de captura esquemática y uso de código de VHDL.

5.5.1 DISEÑO DE CIRCUITOS ARITMÉTICOS CON EL USO DE CAPTURA ESQUEMÁTICA

Una forma obvia de diseñar un circuito aritmético mediante captura esquemática es trazar un esquema que contenga las compuertas lógicas necesarias. Por ejemplo, para crear un sumador de n bits, primero podríamos dibujar un esquema que represente un sumador completo. Luego podría crearse un sumador de n bits con acarreo en cascada trazando un esquema de mayor nivel que conecte las n *instancias* del sumador completo. Un esquema jerárquico creado de esta forma se parecería al circuito de la figura 5.6. También podríamos usar este método para crear un circuito sumador/restador, como el bosquejado en la figura 5.13.

El problema principal de este enfoque es que resulta engorroso, sobre todo cuando el número de bits es grande. Ello se torna aún más evidente si consideramos la creación de un esquema para un sumador con acarreo de adelanto. Como expusimos en la sección 5.4.1, los circuitos de acarreo en cada etapa del sumador con acarreo de adelanto se vuelven cada vez más complejos. Por ende, es preciso trazar un esquema separado por cada una de sus etapas. Un mejor enfoque para crear circuitos aritméticos mediante captura esquemática es usar subcircuitos predefinidos.

En la sección 2.9.1 dijimos que las herramientas de captura esquemática ofrecen una biblioteca de símbolos gráficos que representan compuertas lógicas básicas. Tales compuertas sirven para crear esquemas de circuitos relativamente simples. Además de las compuertas básicas, la

mayor parte de las herramientas de captura esquemática también ofrece una biblioteca de circuitos de uso común, como los sumadores. Cada circuito se ofrece como un módulo que puede importarse en un esquema y utilizarse como parte de un circuito más grande. En algunos sistemas CAD, los módulos se llaman *macrofunciones* o *megafunciones*.

Hay dos tipos principales de macrofunciones: dependientes de la tecnología e independientes de ella. Una *macrofunción dependiente de la tecnología* está diseñada para adaptarse a tipos específicos de chip. Por citar un caso, en la sección 5.4.1 describimos una expresión para un sumador con acarreo de adelanto diseñado para satisfacer una restricción en la carga de entrada de compuertas de cuatro entradas. Una macrofunción que implemente esta expresión sería de tecnología específica. Una *macrofunción independiente de la tecnología* puede implementarse en cualquier tipo de chip. La macrofunción para un sumador que represente diferentes circuitos para distintos tipos de chips es independiente de la tecnología.

Un buen ejemplo de biblioteca de macrofunciones es la *Library of Parameterized Modules* (*LPM*, biblioteca de módulos parametrizados) incluida como parte del sistema CAD Quartus II. Cada módulo de la biblioteca es independiente de la tecnología. Además, está *parametrizado*, lo que significa que puede usarse de varias formas. Por ejemplo, la biblioteca LPM incluye un módulo sumador de *n* bits, llamado *lpm_add_sub*.

En la figura 5.20 se presenta una ilustración esquemática de la capacidad del módulo *lpm_add_sub*, el cual tiene varios parámetros asociados que se configuran mediante las herramientas CAD. Los dos parámetros más importantes para los propósitos de nuestra explicación se llaman LPM_WIDTH y LPM_REPRESENTATION. El parámetro LPM_WIDTH especifica el número de bits, *n*, en el sumador. El parámetro LPM_REPRESENTATION especifica si se usan enteros con o sin signo. Esto sólo afecta la parte del módulo que determina cuándo ocurre desbordamiento aritmético. Para el esquema mostrado, LPM_WIDTH = 16, y se usan números con signo.

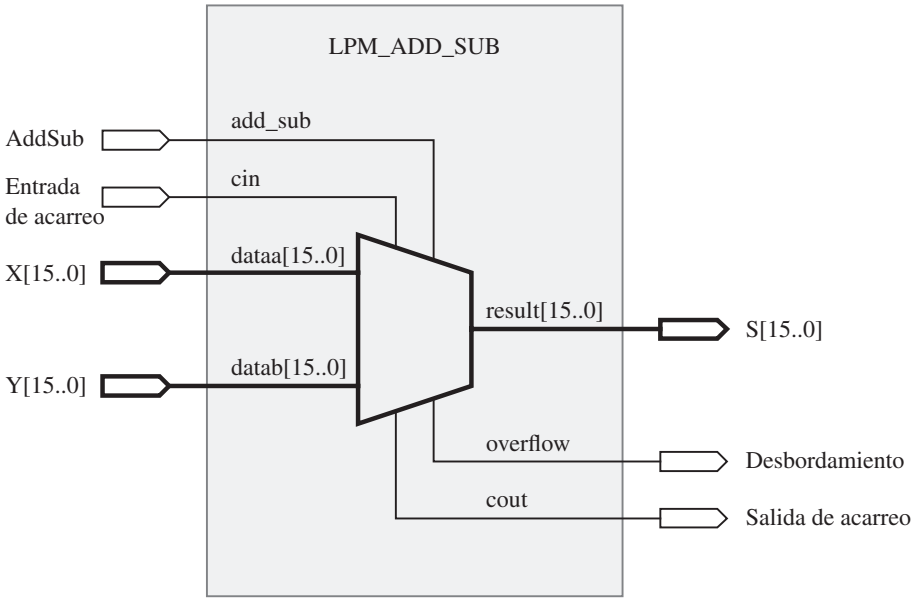


Figura 5.20 Esquema que usa un módulo LPM sumador/restador.

El módulo puede realizar sumas o restas, determinadas por la entrada *add_sub*. Por ende, el módulo representa un circuito sumador/restador, como el mostrado en la figura 5.13.

Los números que el módulo *lpm_add_sub* sumará se conectan a las terminales llamadas *dataa*[15..0] y *datab*[15..0]. Los corchetes en estos nombres significan que representan números multibit. En el esquema, *dataa* y *datab* se conectan a las señales de entrada de 16 bits *X*[15..0] y *Y*[15..0]. El significado de la sintaxis *X*[15..0] es que la señal *X* representa 16 bits, llamados *X*[15], *X*[14], . . . , *X*[0]. El módulo *lpm_add_sub* produce la suma en la terminal llamada *result*[15..0], que se conecta a la salida *S*[15..0]. En la figura 5.20 también se muestra que la LPM soporta una entrada de acarreo, así como salidas de acarreo y desbordamiento.

Para valorar la eficacia de la LPM, el módulo *lpm_add_sub* se configura para realizar un sumador de 16 bits que calcule las salidas suma, acarreo y desbordamiento; esto significa que las señales *add_sub* y *cin* no se necesitan. Usamos herramientas CAD para implementar este circuito en un chip FPGA, y simulamos su rendimiento. El diagrama de tiempo resultante se muestra en la figura 5.21, que es una pantalla del simulador de tiempo. Los valores de las señales de 16 bits, *X*, *Y* y *S* se muestran en la salida de simulación como números hexadecimales. Al comienzo de la simulación, *X* y *Y* se establecen en 0000. Después de 50 ns, *Y* cambia a 0001, lo que ocasiona que *S* cambie a 0001. El siguiente cambio en las entradas ocurre a 150 ns, cuando *X* cambia a 3FFF. Para producir la nueva suma, que es 4000, el sumador debe esperar a que sus señales de acarreo caigan en cascada de la primera a la última etapas. Esto se advierte en la salida de la simulación como una secuencia de cambios rápidos en el valor de *S*, y posteriormente se estabiliza en la suma correcta. Nótese que la línea de referencia del simulador, la línea vertical gruesa en la figura, muestra que la suma correcta se produce 160.93 ns desde el principio de la simulación. Puesto que el cambio en las entradas ocurrió a 150 ns, el sumador demora 160.93 – 150 = 10.93 ns para calcular la suma. A 250 ns, *X* cambia a 7FFF, lo que ocasiona que la suma sea 8000. Esta suma es demasiado grande para un número con signo de 16 bits; por tanto, *Overflow* se hace 1 para indicar el desbordamiento aritmético.

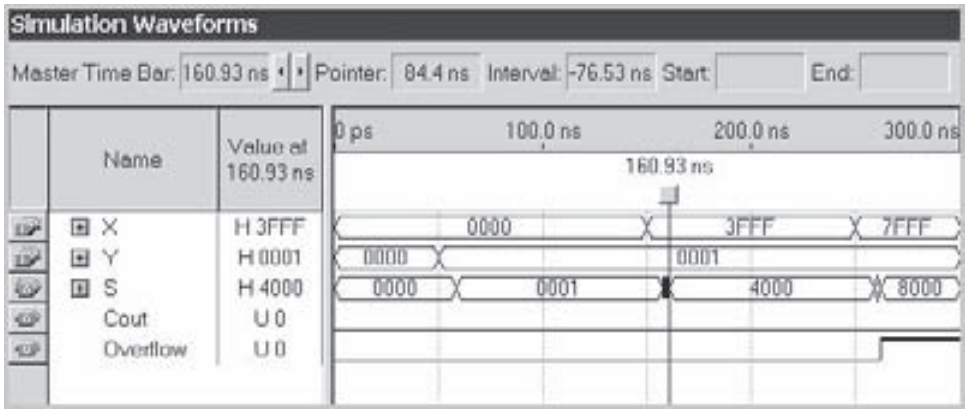


Figura 5.21 Resultados de simulación para el sumador de la LPM.

5.5.2 DISEÑO DE CIRCUITOS ARITMÉTICOS CON VHDL

En la sección 5.5.1 dijimos que una forma obvia de crear un sumador de *n* bits es trazar un esquema jerárquico que contenga *n* sumadores completos. Este enfoque también puede seguirse con VHDL: primero se crea una entidad de VHDL para un sumador completo y luego una entidad de nivel superior que use cuatro *instancias* del sumador completo. Como primer intento en el diseño de circuitos aritméticos mediante VHDL mostraremos cómo escribir el código jerárquico para un sumador con acarreo en cascada.

En la figura 5.22 se presenta el código completo para una entidad de sumador completo. Tiene las entradas *Cin*, *x* y *y*, y produce las salidas *s* y *Cout*. La suma, *s*, y el acarreo de salida, *Cout*, se describen mediante ecuaciones lógicas.

Ahora debemos crear una entidad de VHDL separada para el sumador con acarreo en cascada, que use la entidad *fulladd* como un subcircuito. En la figura 5.23 se muestra un método para hacerlo. En ella se proporciona el código para una entidad sumador con acarreo en cascada de cuatro bits, llamada *adder4*. Uno de los números de cuatro bits por sumar se representa mediante las cuatro señales x_3, x_2, x_1, x_0, y el otro número con y_3, y_2, y_1, y_0. La suma se representa s_3, s_2, s_1, s_0.

Obsérvese que el cuerpo arquitectónico tiene el nombre *Structure*, el cual se eligió porque el estilo del código en el que se describe un circuito de forma jerárquica, mediante la conexión en conjunto de los subcircuitos, usualmente se llama *estructural*. En ejemplos previos de código de VHDL, todas las señales usadas se declararon como puertos en la declaración de entidad. Como se muestra en la figura 5.23, las señales también pueden declarar antes de la palabra clave BEGIN en el cuerpo arquitectónico. Las tres señales declaradas, denominadas c_1, c_2 y c_3, se utilizan como señales de acarreo provenientes de las tres primeras etapas del sumador. La instrucción siguiente se llama de *declaración de componentes*. Utiliza una sintaxis similar a la de una declaración de entidad, y permite que la entidad *fulladd* se use como componente (subcircuito) en el cuerpo arquitectónico.

El sumador de cuatro bits de la figura 5.23 se describe empleando cuatro instrucciones de *instanciación*. Cada una de ellas comienza con un *nombre de instancia*, que puede ser cualquier nombre legal en VHDL, seguido por el signo de dos puntos. Los nombres deben ser únicos. La etapa menos significativa del sumador se llama *stage0*, y la más significativa *stage3*. Después

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y   : IN     STD_LOGIC ;
           s, Cout     : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

Figura 5.22 Código de VHDL para el sumador completo.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin            : IN   STD_LOGIC ;
           x3, x2, x1, x0 : IN   STD_LOGIC ;
           y3, y2, y1, y0 : IN   STD_LOGIC ;
           s3, s2, s1, s0 : OUT  STD_LOGIC ;
           Cout           : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN   STD_LOGIC ;
               s, Cout   : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figura 5.23 Código de VHDL para un sumador de cuatro bits.

de los dos puntos sigue el nombre del componente, *fulladd*, y luego la palabra clave PORT MAP (mapa de puertos). Entonces se hace una lista con los nombres de las señales en la entidad *adder4* que se conectarán a cada puerto de entrada y salida en el componente *fulladd*. Obsérvese que en las primeras tres instrucciones de instanciación, las señales se enumeran en el mismo orden que en la instrucción de declaración COMPONENT de *fulladd*: *Cin*, *x*, *y*, *s*, *Cout*. Pueden enumerarse en otro orden especificando explícitamente cuál de ellas se conectará a qué puerto en el componente. Un ejemplo de este estilo se muestra para la instancia *stage3*. Este estilo de instanciación de componente se conoce como *asociación por nombre* en la terminología de VHDL, mientras que el estilo usado en las otras tres instancias se denomina *asociación por posición*. Nótese que para la instancia *stage3* el nombre de señal *Cout* se usa tanto para el nombre del puerto componente como para el de la señal en la entidad *adder4*. Esto no causa problema al compilador de VHDL, pues el nombre del puerto componente siempre es el del lado izquierdo de los caracteres =>.

Los nombres de señal asociados con cada instancia del componente *fulladd* especifican de manera implícita cómo se conectan los sumadores completos. Por ejemplo, la salida de acarreo de la instancia *stage0* se conecta a la entrada de acarreo de la instancia *stage1*. Cuando el código de la figura 5.23 se analiza mediante el compilador de VHDL, automáticamente busca el código que

debe emplear para el componente *fulladd* presentado en la figura 5.22. El circuito sintetizado tiene la misma estructura que el de la figura 5.6.

Estilo alternativo de código

En la figura 5.23, una instrucción de declaración de componente para la entidad *fulladd* se incluye en la arquitectura *adder4*. Un enfoque alternativo consiste en colocar la instrucción de declaración de componente en un *paquete* de VHDL. En general, un paquete permite que los constructores de VHDL se definan en un archivo de código fuente y luego se usen en otros archivos semejantes. Dos ejemplos de constructores que con frecuencia se colocan en un paquete son las declaraciones de tipo datos y las de componentes.

Ya vimos un ejemplo de cómo utilizar un paquete para un tipo de datos. En el capítulo 4 presentamos el paquete llamado *std_logic_1164*, que define el tipo de señal STD_LOGIC. Recuérdese que para acceder a este paquete el código de VHDL debe incluir las instrucciones

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

Las cuales aparecen en las figuras 5.22 y 5.23 porque el tipo STD_LOGIC se usa en el código. La primera proporciona acceso a la biblioteca llamada *ieee*. Como explicamos en la sección 4.12, la biblioteca representa la ubicación, o *directorio*, en el sistema de archivos de la computadora donde se almacena el paquete *std_logic_1164*.

El código de la figura 5.24 define el paquete llamado *fulladd_package*. Este código puede almacenarse en un archivo de código fuente de VHDL por separado o incluirse en el mismo archivo de código fuente utilizado para almacenar el código de la entidad *fulladd* mostrado en la figura 5.22. La sintaxis de VHDL exige que la declaración de paquete tenga sus propias cláusulas LIBRARY y USE; por tanto, se incluyen en el código. Dentro del paquete, la entidad *fulladd* se declara como un COMPONENT. Cuando este código se compila se crea el paquete *fulladd_package* y se almacena en el directorio de trabajo donde se guarda el código.

Entonces toda entidad de VHDL puede usar el componente *fulladd* como un subcircuito para utilizar el paquete *fulladd_package*. Al paquete se accede empleando las dos instrucciones siguientes

```
LIBRARY work;
USE work.fulladd_package.all ;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figura 5.24 Declaración de un paquete.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin            : IN   STD_LOGIC ;
           x3, x2, x1, x0 : IN   STD_LOGIC ;
           y3, y2, y1, y0 : IN   STD_LOGIC ;
           s3, s2, s1, s0 : OUT  STD_LOGIC ;
           Cout           : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figura 5.25 Una forma diferente de especificar un sumador de cuatro bits.

La biblioteca llamada *work* representa el directorio de trabajo donde se almacena el código de VHDL que define el paquete. En realidad, esta instrucción no es necesaria, ya que el compilador de VHDL siempre tiene acceso al directorio de trabajo.

En la figura 5.25 se muestra cómo volver a escribir el código de la figura 5.23 para usar el *fulladd_package*. El código es el mismo que el de la figura 5.23 con dos excepciones: se agrega la cláusula adicional USE y en la arquitectura se borra la instrucción de declaración de componente. Los circuitos sintetizados de las dos versiones del código son idénticos.

En las figuras 5.23 y 5.25, cada una de las entradas de cuatro bits y la salida de cuatro bits del sumador se representan con señales de un solo bit. Un estilo más de código más práctico es usar señales multibit para representar los números.

5.5.3 REPRESENTACIÓN DE NÚMEROS EN CÓDIGO DE VHDL

Igual que un número se representa en un circuito lógico como señales sobre varios cables, un número se indica en código de VHDL como un objeto de datos SIGNAL multibit. El siguiente es un ejemplo de una señal multibit

```
SIGNAL C  : STD_LOGIC_VECTOR (1 TO 3) ;
```

El tipo de datos STD_LOGIC_VECTOR representa un arreglo lineal de objetos de datos STD_LOGIC. En VHDL se dice que STD_LOGIC_VECTOR es un subtipo de STD_LOGIC. Hay un subtipo similar, llamado BIT_VECTOR, que corresponde al tipo BIT usado en la sec-

ción 2.10.2. La precedente declaración SIGNAL define C como una señal STD_LOGIC de tres bits. Puede usarse en código de VHDL como una cantidad de tres bits simplemente empleando el nombre *C*, o de otro modo cada bit individual se puede referir por separado mediante los nombres *C*(1), *C*(2) y *C*(3). La sintaxis 1 TO 3 en la instrucción de declaración especifica que el bit más significativo en *C* se llama *C*(1) y el menos significativo *C*(3). A *C* puede asignársele un valor de señal de tres bits del modo siguiente:

```
C <= "100" ;
```

El valor de tres bits se denota con comillas, en lugar de los apóstrofos que se utilizan para valores de un bit, como en '1' o '0'. La instrucción de asignación resulta en $C(1) = 1$, $C(2) = 0$ y $C(3) = 0$. La numeración de los bits en la señal *C*, con el índice más alto usado para el menos significativo, es una forma natural de representar señales que sólo están agrupadas por conveniencia pero que no representan un número. Por ejemplo, este esquema de numeración sería una forma adecuada de declarar las tres señales de acarreo llamadas c_1, c_2 y c_3 en la figura 5.25. Sin embargo, cuando se emplea una señal multibit para representar un número binario es más lógico numerar los bits de forma opuesta, con el índice más alto para el bit más significativo. Para este propósito, VHDL ofrece una segunda forma de declarar una señal multibit:

```
SIGNAL X : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

Esta instrucción define *X* como una señal STD_LOGIC_VECTOR de cuatro bits. La sintaxis 3 DOWNTO 0 especifica que el bit más significativo en *X* se llama *X*(3) y el menos significativo es *X*(0). Este esquema es una forma más natural de numerar los bits si se usará *X* en el código de VHDL para representar un número binario porque el índice de cada bit corresponde a su posición en el número. La instrucción de asignación

```
X <= "1100" ;
```

resulta en $X(3) = 1$, $X(2) = 1$, $X(1) = 0$ y $X(0) = 0$.

En la figura 5.26 se muestra cómo escribir el código de la figura 5.25 para usar señales multibit. Las entradas de datos son las señales de cuatro bits *X* y *Y*, y la salida suma es la señal de cuatro bits *S*. Las señales intermedias de acarreo se declaran en la arquitectura como la señal de tres bits *C*.

El uso de código de VHDL jerárquico para definir grandes circuitos aritméticos puede ser engorroso. Por ello, los circuitos aritméticos se implementan de otra forma en VHDL, con instrucciones de asignación aritmética y señales multibit.

5.5.4 INSTRUCCIONES DE ASIGNACIÓN ARITMÉTICA

Si se definen las señales siguientes

```
SIGNAL X, Y, S : STD_LOGIC_VECTOR (15 DOWNTO 0) ;
```

entonces la instrucción de asignación aritmética

```
S <= X + Y;
```

representa un sumador de 16 bits.

Además del operador +, usado para la suma, VHDL ofrece otros operadores aritméticos. En la tabla A.1 del Apéndice A hay una lista de ellos. En la figura 5.27 se ofrece todo el código

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figura 5.26 Sumador de cuatro bits definido mediante señales multibit.

de VHDL que incluye la instrucción anterior. El paquete *std_logic_1164* no especifica que las señales STD_LOGIC puedan utilizarse con operadores aritméticos. El segundo paquete incluido en el código, *std_logic_signed*, permite que las señales se usen de esta forma. Cuando el compilador de VHDL traduce el código de la figura genera un circuito sumador para implementar el operador +. Cuando se emplea el sistema CAD Quartus II, el sumador que ocupa el compilador es en realidad el módulo *lpm_add_sub* mostrado en la figura 5.20. El compilador automáticamente establece los parámetros para el módulo, de modo que representa un sumador de 16 bits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( X, Y  : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figura 5.27 Código de VHDL para un sumador de 16 bits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S                : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0'& X) + Y + Cin ;
    S<= Sum(15 DOWNTO 0) ;
    Cout<= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figura 5.28 Sumador de 16 bits de la figura 5.27 con señales de acarreo y desbordamiento.

El código de la figura 5.27 no incluye señales de salida o entrada de acarreo. Además, no ofrece la señal de desbordamiento aritmético. En la figura 5.28 se presenta un modo en que es posible agregar estas señales. La señal de 17 bits llamada *Sum* se define en la arquitectura. El bit adicional, *Sum*(16), se usa para la salida de acarreo de la posición de bit 15 en el sumador. La instrucción empleada para asignar la suma de *X*, *Y* y la entrada de acarreo, *Cin*, a la señal *Sum* recurre a una sintaxis inusual. El significado del término entre paréntesis, ('0' & X), es que un 0 se concatena a la señal de 16 bits *X* para crear una señal de 17 bits. En VHDL, & se llama operador de *concatenación*. El lector no debe confundir este significado de & con el más tradicional de otros lenguajes de descripción de hardware donde es el operador lógico AND. La razón por la que el operador de concatenación se necesita en la figura 5.28 es que VHDL requiere que al menos uno de los operandos de una expresión aritmética tenga el mismo número de bits que el resultado. Puesto que *Sum* es un operando de 17 bits, entonces por lo menos *X* o *Y* deben modificarse para convertirse en un número de 17 bits.

Otro detalle que ha de observarse en la figura es la instrucción

```
S <= Sum(15 DOWNTO 0) ;
```

Que asigna los 16 bits menos significativos de *Sum* a la suma de salida *S*. La instrucción siguiente asigna la salida de acarreo de la suma, *Sum*(16), a la señal de salida de acarreo, *Cout*. La expresión para el desbordamiento aritmético se definió en la sección 5.3.5 como $c_{n-1} \oplus c_n$. En este caso, c_n corresponde a *Sum*(16), pero no hay forma directa de acceder a c_{n-1}, que es la salida de acarreo de la posición de bit 14. El lector debe comprobar que la expresión $X(15) \oplus Y(15) \oplus Sum(15)$ corresponde a c_{n-1}.

Dijimos que el compilador de VHDL puede generar un circuito sumador para implementar el operador +, y que el sistema Quartus II en realidad usa el módulo *lpm_add_sub* para ello.

Para redondear el tema también debemos mencionar que el módulo *lpm_add_sub* puede instanciarse directamente en código de VHDL, de forma similar a como se instanció el componente *fulladd* en la figura 5.23. En la sección A.6 del Apéndice A se da un ejemplo.

El código de la figura 5.28 usa el paquete *std_logic_signed* para permitir que las señales STD_LOGIC se empleen con operadores aritméticos. El paquete *std_logic_signed* en realidad utiliza otro paquete, *std_logic_arith*, que define dos tipos de datos: SIGNED y UNSIGNED, para usarlos en circuitos aritméticos que emplean números con y sin signo. Estos tipos de datos son los mismos que el tipo STD_LOGIC_VECTOR; cada uno es un arreglo de señales STD_LOGIC. El código de la figura 5.28 puede escribirse para usar directamente el paquete *std_logic_arith* como se muestra en la figura 5.29. Las señales multibit *X*, *Y*, *S* y *Sum* tienen el tipo SIGNED. De otro modo, el código es idéntico al de la figura 5.28 y resulta en el mismo circuito.

Es una elección arbitraria si se usa el paquete *std_logic_signed* y las señales STD_LOGIC_VECTOR, como en la figura 5.28, o el paquete *std_logic_arith* y las señales SIGNED, como en la 5.29. Para usar números sin signo también hay dos opciones. Se puede utilizar el paquete *std_logic_unsigned* con señales STD_LOGIC_VECTOR o el paquete *std_logic_arith* con señales UNSIGNED. Para nuestro código de ejemplo de las figuras 5.28 y 5.29, se generaría el mismo circuito si se suponen números con o sin signo. Pero para números sin signo no debemos producir una salida *Overflow* por separado, ya que la salida de acarreo representa el desbordamiento aritmético para los números sin signo.

Antes de dejar la explicación de las instrucciones aritméticas en VHDL debemos mencionar otra señal de tipo de datos que sirve para la aritmética. La instrucción siguiente define la señal *X*

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   SIGNED(15 DOWNTO 0) ;
           S                : OUT  SIGNED(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16);
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figura 5.29 Uso del paquete aritmético.

```
ENTITY adder16 IS
    PORT ( X, Y  : IN   INTEGER RANGE -32768 TO 32767 ;
           S     : OUT  INTEGER RANGE -32768 TO 32767 ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figura 5.30 El sumador de 16 bits de la figura 5.27 con señales INTEGER.

como un entero (INTEGER)

```
SIGNAL X : INTEGER RANGE −32768 TO 32767
```

Para un objeto de datos INTEGER, el número de bits no se especifica explícitamente. En vez de ello, se especifica el *intervalo* de números que se representarán. Para un entero con signo de 16 bits, el intervalo de números representables es −32768 a 32767. Un ejemplo del uso del tipo de datos INTEGER en el código correspondiente a la figura 5.27 se muestra en la figura 5.30. Ahí no aparecen cláusulas LIBRARY o USE, porque el tipo INTEGER está predefinido en el VHDL estándar. Aunque el código de la figura es directo, es más difícil modificarlo para incluir las señales de acarreo y la salida de desbordamiento que se muestran en las figuras 5.28 y 5.29. El método que usamos, en el que los bits provenientes de la señal *Sum* sirvieron para definir las señales de salida de acarreo y de desbordamiento aritmético, no puede emplearse para objetos INTEGER.

5.6 MULTIPLICACIÓN

Antes de abordar el tema general de la multiplicación, cabe señalar que un número binario, B, puede multiplicarse por 2 simplemente agregando un cero a la derecha de su bit menos significativo. Esto efectivamente mueve todos los bits de B a la izquierda, y se dice que B se corre a la izquierda una posición de bit. Por ende, si $B = b_{n-1}b_{n-2}\cdots b_1b_0$, entonces $2 \times B = b_{n-1}b_{n-2}\cdots b_1b_00$. (En la sección 5.2.3 ya empleamos este hecho.) De manera similar, un número se multiplica por 2^k corriéndolo a la izquierda k posiciones de bit. Esto es cierto tanto para números con signo como para los que no lo llevan.

También habremos de considerar lo que ocurre si un número binario se corre a la derecha k posiciones de bit. De acuerdo con la representación numérica posicional, esta acción divide el número entre 2^k. Para números sin signo, el corrimiento explica la adición de k ceros a la izquierda del bit más significativo. Por ejemplo, si B es un número sin signo, entonces $B \div 2 = 0b_{n-1}b_{n-2}\cdots b_2b_1$. Nótese que el bit b_0 se pierde cuando se corre a la derecha. En números con signo es preciso conservar éste, lo cual se hace corriendo los bits a la derecha y llenando desde la izquierda con el valor del bit del signo. Por tanto, si B es un número con signo, entonces $B \div 2 = b_{n-1}b_{n-1}b_{n-2}\cdots b_2b_1$. Por ejemplo, si $B = 011000 = (24)_{10}$ entonces; $B \div 2 = 001100 = (12)_{10}$ y $B \div 4 = 000110 = (6)_{10}$. De manera similar, si $B = 101000 = -(24)_{10}$,

entonces $B \div 2 = 110100 = -(12)_{10}$ y $B \div 4 = 111010 = -(6)_{10}$. El lector también debe observar que cuanto más pequeño sea el número positivo, más 0 quedarán a la izquierda del primer 1, mientras que para los números negativos habrá más 1 a la izquierda del primer 0.

Ahora podemos centrar la atención en la tarea general de la multiplicación. Dos números binarios pueden multiplicarse mediante el mismo método empleado para los números decimales. Limitaremos la explicación a la multiplicación de números sin signo. En la figura 5.31*a* se muestra cómo se realiza la multiplicación manualmente, con números de cuatro bits. Cada bit multiplicador se examina de derecha a izquierda. Si un bit es igual a 1, se agrega una versión apropiadamente corrida del multiplicando para formar un *producto parcial*. Si el bit multiplicador es igual a 0, entonces nada se suma. La suma de todas las versiones corridas del multiplicando es el producto deseado. Nótese que éste ocupa ocho bits.

El mismo esquema sirve para diseñar un circuito multiplicador. En este ejemplo conservaremos los números de cuatro bits para mantener simple el análisis. Sean $M = m_3m_2m_1m_0$, $Q = q_3q_2q_1q_0$ y $P = p_7p_6p_5p_4p_3p_2p_1p_0$ el multiplicando, el multiplicador y el producto, respectivamente. Una forma simple de implementar el esquema de la multiplicación es usar un enfoque secuencial,

Multiplicando M	(14)	1 1 1 0
Multiplicador Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Producto P	(154)	1 0 0 1 1 0 1 0

a) Multiplicación a mano

Multiplicando M	(11)	1 1 1 0
Multiplicador Q	(14)	× 1 0 1 1
Producto parcial 0		1 1 1 0
		+ 1 1 1 0
Producto parcial 1		1 0 1 0 1
		+ 0 0 0 0
Producto parcial 2		0 1 0 1 0
		+ 1 1 1 0
Producto P	(154)	1 0 0 1 1 0 1 0

b) Multiplicación para implementación en hardware

Figura 5.31 Multiplicación de números sin signo.

donde un sumador de ocho bits se utilice para calcular productos parciales. Como primer paso, se examina el bit q_0. Si $q_0 = 1$, entonces M se suma al producto parcial inicial, que se inicializa en 0. Si $q_0 = 0$, entonces se suma 0 al producto parcial. A continuación se examina q_1. Si $q_1 = 1$, entonces se suma el valor $2 \times M$ al producto parcial. El valor $2 \times M$ se crea con sólo correr M una posición de bit a la izquierda. De manera similar, $4 \times M$ se suma al producto parcial si $q_2 = 1$, y $8 \times M$ se suma si $q_3 = 1$. En el capítulo 10 demostraremos cómo implementar tal circuito.

Este enfoque secuencial conduce a un circuito relativamente lento, sobre todo porque usa un solo sumador de ocho bits para realizar las sumas necesarias para generar los productos parciales y final. Es posible obtener un circuito mucho más rápido si se emplean varios sumadores para calcular los productos parciales.

5.6.1 ARREGLO MULTIPLICADOR PARA NÚMEROS SIN SIGNO

En la figura 5.31*b* se indica cómo llevar a cabo una multiplicación usando varios sumadores. En cada paso se utiliza un sumador de cuatro bits para calcular el nuevo producto parcial. Nótese que a medida que el cálculo avanza, los bits menos significativos no se afectan por las sumas subsecuentes; por tanto, se pueden pasar directamente al producto final, como se indica mediante las flechas gris oscuro. Desde luego, dichos bits también son parte de los productos parciales.

Es posible diseñar un circuito multiplicador más rápido con un arreglo estructural de organización semejante al de la figura 5.31*b*. Considérese un ejemplo de 4×4, donde el multiplicando y el multiplicador son $M = m_3m_2m_1m_0$ y $Q = q_3q_2q_1q_0$, respectivamente. El producto parcial 0, $PP0 = pp0_3\ pp0_2\ pp0_1\ pp0_0$, puede generarse usando la AND de q_0 con cada bit de M. Por ende

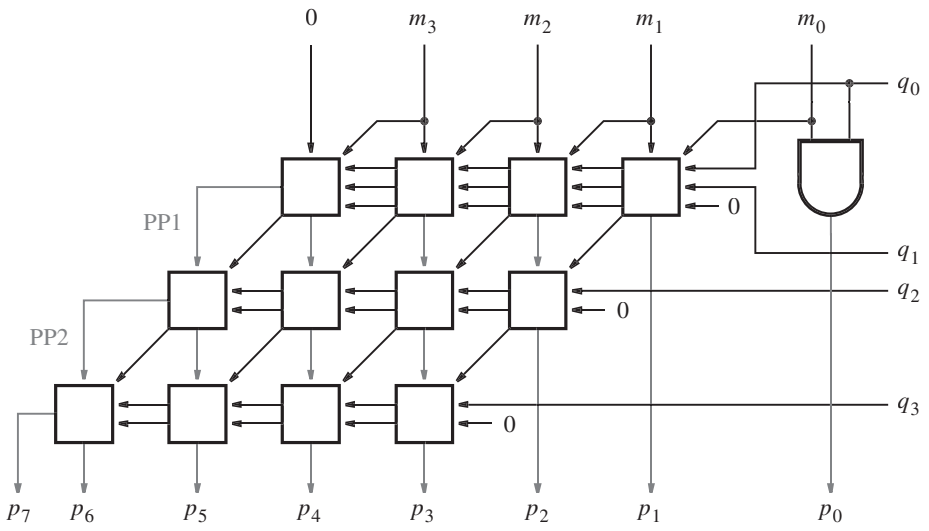
$$PP0 = m_3q_0\ \ m_2q_0\ \ m_1q_0\ \ m_0q_0$$

El producto parcial 1, $PP1$, se genera usando la AND de q_1 con M y sumándola a $PP0$ como sigue

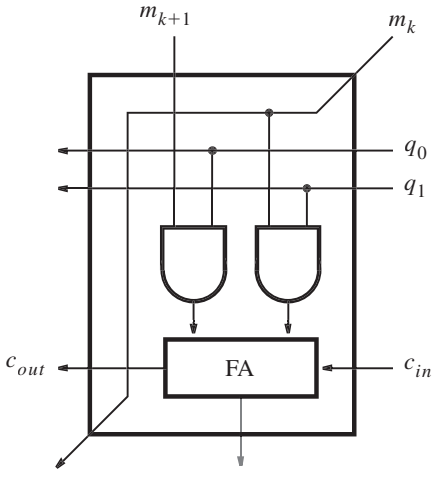
$PP0$:		0	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$
	+	m_3q_1	m_2q_1	m_1q_1	m_0q_1	0
$PP1$:		$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$

De manera similar, el producto parcial 2, $PP2$, se genera empleando la AND de q_2 con M y sumándola a $PP1$, etcétera.

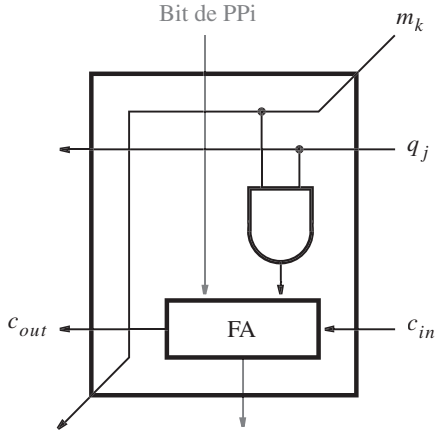
Un circuito que implemente las operaciones anteriores se organiza en un arreglo, como se muestra en la figura 5.32*a*. Hay dos tipos de bloques en el arreglo. En el inciso (*b*) de la figura se presentan los detalles de los bloques de la fila superior, y en el inciso (*c*) se muestra el bloque utilizado en la segunda y tercera filas. Obsérvese que las versiones corridas del multiplicando se ofrecen al enrutar las señales m_k diagonalmente de un bloque a otro. El sumador completo incluido en cada bloque implementa un sumador con acarreo en cascada para generar cada producto parcial. Es posible diseñar multiplicadores incluso más veloces utilizando otros tipos de sumadores [1].



a) Estructura del circuito



b) Bloque de la fila superior



c) Bloque de las dos filas inferiores

Figura 5.32 Circuito multiplicador de 4 × 4.

5.6.2 MULTIPLICACIÓN DE NÚMEROS CON SIGNO

La multiplicación de números sin signo ilustra los aspectos centrales relativos al diseño de circuitos multiplicadores. La multiplicación de números con signo es un tanto más compleja.

Si el operando multiplicador es positivo, es posible usar esencialmente el mismo esquema que para los números sin signo. Por cada bit del operando multiplicador que sea igual a 1 hay

que sumar al producto parcial una versión adecuadamente corrida del multiplicando. El multiplicando puede ser o positivo o negativo.

Puesto que las versiones corridas del multiplicando se suman a los productos parciales es importante garantizar que los números implicados estén representados de manera correcta. Por ejemplo, si los dos bits más a la derecha del multiplicador son iguales a 1, entonces la primera suma debe producir el producto parcial $PP1 = M + 2M$, donde M es el multiplicando. Si $M = m_{n-1}m_{n-2} \cdots m_1m_0$, entonces $PP1 = m_{n-1}m_{n-2} \cdots m_1m_0 + m_{n-1}m_{n-2} \cdots m_1m_00$. El sumador que realiza esta suma comprende circuitos que suman dos operandos de igual longitud. Puesto que el corrimiento del multiplicando a la izquierda para generar $2M$ resulta en que uno de los operandos tenga $n + 1$ bits, la suma requerida debe llevarse a cabo usando el segundo operando, M, representado también como un número de $(n + 1)$ bits. Un número con signo de n bits se representa como un número de $(n + 1)$ bits replicando el bit del signo como el nuevo bit más a la izquierda. Por tanto, $M = m_{n-1}m_{n-2} \cdots m_1m_0$ se representa empleando $(n + 1)$ bits como $M = m_{n-1}m_{n-1}m_{n-2} \cdots m_1m_0$. El valor de un número positivo no cambia si se añaden 0 como los bits más significativos; el valor de un número negativo no cambia si se añaden 1 como los bits más significativos. Tal replicación del bit del signo se llama *extensión de signo*.

Cuando a un producto parcial se suma una versión corrida del multiplicando hay que evitar el desbordamiento. Por consiguiente, el nuevo producto parcial debe ser más grande en un bit. En la figura 5.33*a* se ilustra el proceso de multiplicar dos números positivos. Los bits de signo extendido se muestran en gris oscuro. En el inciso (*b*) de la figura se supone un multiplicando negativo. Nótese que el producto resultante tiene $2n$ bits en ambos casos.

Para un operando multiplicador negativo es posible convertir tanto el multiplicador como el multiplicando en sus complementos a 2 porque ello no cambiará el valor del resultado. Entonces puede usarse el esquema para un multiplicador positivo.

Hemos presentado un esquema hasta cierto punto simple para multiplicar números con signo. Hay otras técnicas más eficientes, pero también más complejas. No las seguiremos, pero el lector interesado puede consultar la referencia [1].

Ya expusimos los circuitos que realizan suma, resta y multiplicación. Otra operación aritmética necesaria en los sistemas de cómputo es la división. Los circuitos que efectúan divisiones son más complejos; en el capítulo 10 presentaremos un ejemplo. En los libros que tratan acerca de la organización de las computadoras se analizan varias técnicas para realizar divisiones, y pueden encontrarse en las referencias [1, 2].

5.7 OTRAS REPRESENTACIONES NUMÉRICAS

En las secciones previas empleamos enteros binarios denotados en representación numérica posicional. En los sistemas digitales también se usan otros tipos de números. En esta sección expondremos brevemente otros tres tipos: punto fijo, punto flotante y números decimales codificados en binario.

5.7.1 NÚMEROS CON PUNTO FIJO

Un número con *punto fijo* consta de partes entera y fraccionaria. Puede escribirse en la representación numérica posicional como

Multiplicando M	(+14)	0 1 1 1 0
Multiplicador Q	(+11)	× 0 1 0 1 1
Producto parcial 0		0 0 0 1 1 1 0
		+ 0 0 1 1 1 0
Producto parcial 1		0 0 1 0 1 0 1
		+ 0 0 0 0 0 0
Producto parcial 2		0 0 0 1 0 1 0
		+ 0 0 1 1 1 0
Producto parcial 3		0 0 1 0 0 1 1
		+ 0 0 0 0 0 0
Producto P	(+154)	0 0 1 0 0 1 1 0 1 0

a) Multiplicando positivo

Multiplicando M	(−14)	1 0 0 1 0
Multiplicador Q	(+11)	× 0 1 0 1 1
Producto parcial 0		1 1 1 0 0 1 0
		+ 1 1 0 0 1 0
Producto parcial 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0
Producto parcial 2		1 1 1 0 1 0 1
		+ 1 1 0 0 1 0
Producto parcial 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0
Producto P	(−154)	1 1 0 1 1 0 0 1 1 0

b) Multiplicando negativo

Figura 5.33 Multiplicación de números con signo.

$$B = b_{n-1}b_{n-2}\cdots b_1b_0.b_{-1}b_{-2}\cdots b_{-k}$$

El valor del número es

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

La posición del punto de la base (raíz) se supone fija; de ahí el nombre de *número con punto fijo*. Si el punto de la base no se muestra, entonces se asume que está a la derecha del dígito menos significativo, lo que indica que el número es un entero.

Los circuitos lógicos que tratan con números con punto fijo son, en esencia, los mismos que los utilizados para los enteros. No los explicaremos por separado.

5.7.2 NÚMEROS CON PUNTO FLOTANTE

Los números con punto fijo tienen un intervalo señalado por los dígitos significativos usados para representar el número. Por ejemplo, si se utilizan ocho dígitos y un signo para representar enteros decimales, entonces el intervalo de valores que pueden representarse va de 0 a ± 99999999. Si se emplean ocho dígitos para representar una fracción, entonces el intervalo representable va de 0.00000001 a ± 0.99999999. En aplicaciones científicas suele ser necesario lidiar con números muy grandes o muy pequeños. En vez de usar la representación con punto fijo, que requeriría muchos dígitos significativos, es mejor utilizar la representación con punto flotante en la que los números se representan mediante una *mantisa* que comprende los dígitos significativos y un *exponente* de la base *R*. El formato es

$$Mantisa \times R^{Exponente}$$

Con frecuencia, los números se *normalizan*, de modo que el punto de la base se coloca a la derecha del primer dígito distinto de cero, como en 5.234×10^{43} o 6.31×10^{-28}.

La representación binaria con punto flotante la norma el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, por sus siglas en inglés) [3]. En la norma se especifican dos tamaños de formatos: un formato de 32 bits de *precisión sencilla* y un formato de 64 bits de *doble precisión*. Ambos formatos se ilustran en la figura 5.34.

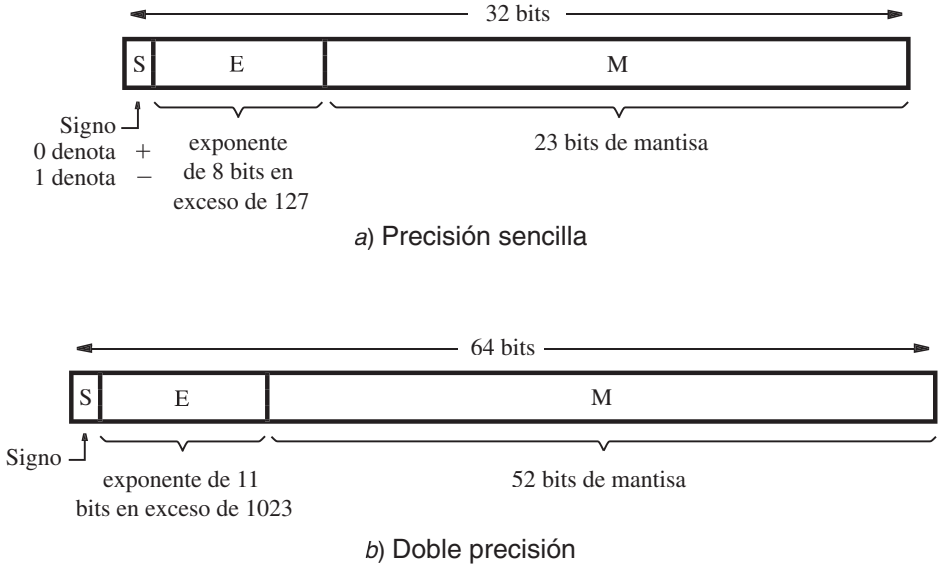


Figura 5.34 Normas del IEEE para formatos con punto flotante.

Formato con punto flotante de precisión sencilla

En la figura 5.34*a* se describe el formato de precisión sencilla. El bit más a la izquierda es el del signo: 0 para números positivos y 1 para negativos. Existe un campo exponente, E, de 8 bits, y un campo mantisa, M, de 23 bits. El exponente es respecto a la base 2. Puesto que es necesario tener capacidad para representar números muy grandes y muy pequeños, el exponente puede ser positivo o negativo. En vez de simplemente usar un número con signo de 8 bits como exponente, lo que permitiría valores de exponente en el intervalo de –128 a 127, la norma del IEEE especifica el exponente en formato de *exceso de 127*. En este formato, el valor 127 se suma al valor del exponente real de modo que

$$Exponente = E - 127$$

De esta forma, E se convierte en un entero positivo. Este formato es práctico para sumar y restar números con punto flotante, ya que el primer paso en estas operaciones supone la comparación de los exponentes para determinar si la mantisa debe correrse adecuadamente para sumar/restar los bits significativos. El intervalo de E va de 0 a 255. Los valores extremo de $E = 0$ y $E = 255$ se toman para denotar el cero exacto y el infinito, respectivamente. Por tanto, el intervalo normal del exponente es -126 a 127, que se representa mediante los valores de E de 1 a 254.

La mantisa se representa mediante 23 bits. La norma del IEEE pide una mantisa normalizada, lo que implica que el bit más significativo siempre es igual a 1. Por ende, no es necesario incluir este bit explícitamente en el campo mantisa. En consecuencia, si M es el vector bit en el campo mantisa, el valor real de la mantisa es 1.M, lo que produce una mantisa de 24 bits. Consecuentemente, el formato de punto flotante de la figura 5.34*a* representa el número

$$\text{Valor} = \pm 1.M \times 2^{E-127}$$

El tamaño del campo mantisa permite la representación de números que tienen la precisión de más o menos siete dígitos decimales. El intervalo del campo exponente, de 2^{-126} a 2^{127}, corresponde aproximadamente a $10^{\pm 38}$.

Formato con punto flotante de doble precisión

En la figura 5.34*b* se muestra el formato de doble precisión, que usa 64 bits. Los campos exponente y mantisa son más grandes. Este formato permite un mayor intervalo y precisión de números. El campo exponente tiene 11 bits y especifica el exponente en formato de *exceso de 1023*, donde

$$Exponente = E - 1023$$

El intervalo de E va de 0 a 2047, pero de nuevo los valores $E = 0$ y $E = 2047$ se usan para indicar el 0 exacto y el infinito, respectivamente. Por tanto, el intervalo normal del exponente va de -1022 a 1023, lo que se representa con los valores de E de 1 a 2046.

El campo mantisa tiene 52 bits. Como se supone que la mantisa está normalizada, su valor real de nuevo es 1.M. Por ello el valor de un número con punto flotante es

$$\text{Valor} = \pm 1.M \times 2^{E-1023}$$

Este formato permite representar números que tienen la precisión de alrededor de 16 dígitos decimales y el intervalo de aproximadamente $10^{\pm 308}$.

Las operaciones aritméticas que usan operandos con punto flotante son mucho más complejas que las operaciones con enteros con signo. Puesto que éste es un terreno más bien especializado, no se ahondará en torno al diseño de circuitos lógicos que puedan realizar tales operaciones. Para una explicación más completa de las operaciones con punto flotante, el lector puede consultar las referencias [1, 2].

5.7.3 REPRESENTACIÓN DECIMAL CODIFICADO EN BINARIO

En los sistemas digitales es posible representar números decimales simplemente codificando cada dígito en forma binaria. Esto se denomina representación *decimal codificado en binario* (*BCD, binary-coded-decimal*). Puesto que existen 10 dígitos para codificar, es preciso usar cuatro bits por cada uno de ellos. Cada dígito se codifica mediante el patrón binario que representa su valor sin signo, como se muestra en la tabla 5.3. Nótese que en BCD sólo se usan 10 de los 16 patrones disponibles, lo que significa que los restantes seis no deben ocurrir en circuitos lógicos que funcionen con operandos BCD; tales patrones usualmente se tratan como condiciones "no importa" en el proceso de diseño. La representación BCD se empleó en algunas de las primeras computadoras, así como en muchas calculadoras de mano. Su principal virtud radica en que ofrece un formato práctico cuando hay que mostrar información numérica en una pantalla orientada a dígitos. Sus inconvenientes son la complejidad de los circuitos que realizan las operaciones aritméticas y el hecho de que se desperdician seis de los posibles patrones de código.

Aun cuando la importancia de la representación BCD ha disminuido, todavía se le encuentra. Para brindar al lector una pista de la complejidad de los circuitos requeridos estudiaremos la suma BCD con cierto detalle.

Suma BCD

La suma de dos dígitos BCD se complica por el hecho de que la suma puede exceder 9, en cuyo caso se tendrá que hacer una corrección. Sean $X = x_3x_2x_1x_0$ y $Y = y_3y_2y_1y_0$ las representaciones de dos dígitos BCD, y sea $S = s_3s_2s_1s_0$ el dígito suma deseado, $S = X + Y$. Obviamente, si $X + Y \leq 9$, entonces la suma es la misma que la suma de dos números binarios sin signo

Tabla 5.3 Dígitos decimales codificados en binario.

Dígito decimal	Código BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

de cuatro bits. Pero si $X + Y > 9$, entonces el resultado requiere dos dígitos BCD. Más aún, la suma de cuatro bits obtenida del sumador de cuatro bits puede ser incorrecta.

En algunos casos es preciso hacer alguna corrección: cuando la suma es mayor que 9 pero no se genera salida de acarreo usando cuatro bits, y cuando la suma es mayor que 15 de modo que se genera salida de acarreo empleando cuatro bits. En la figura 5.35 se ilustran estos casos. En el primero, la suma de cuatro bits produce $7 + 5 = 12 = Z$. Para obtener un resultado BCD correcto hay que generar $S = 2$ y un acarreo de 1. La corrección necesaria es evidente a partir del hecho de que la suma de cuatro bits es un esquema de módulo 16, mientras que la suma decimal es un esquema de módulo 10. Por tanto, puede generarse un dígito decimal correcto sumando 6 al resultado de la suma de cuatro bits siempre que este resultado supere 9. En consecuencia, podemos arreglar el cálculo como sigue

$$Z = X + Y$$

Si $Z \leq 9$, entonces $S = Z$ y salida de acarreo $= 0$

Si $Z > 9$, entonces $X = Z + 6$ y salida de acarreo $= 1$

El segundo ejemplo de la figura 5.35 muestra lo que ocurre cuando $X + Y > 15$. En este caso, los cuatro bits menos significativos de Z representan el dígito 1, lo cual es erróneo. Pero se genera un acarreo, que corresponde al valor 16, que debe tomarse en cuenta. De nuevo, si se suma 6 a la suma intermedia Z se produce la corrección necesaria.

En la figura 5.36 se presenta un diagrama de bloques de un sumador BCD de un dígito basado en este esquema. El bloque que detecta si $Z > 9$ produce una señal de salida, *Adjust* (ajuste), que controla al multiplexor que proporciona la corrección cuando es necesario. Un segundo sumador de cuatro bits genera los bits suma corregidos. Si *Adjust* $= 0$, entonces $S = Z + 0$; si *Adjust* $= 1$, entonces $S = Z + 6$ y salida de acarreo $= 1$.

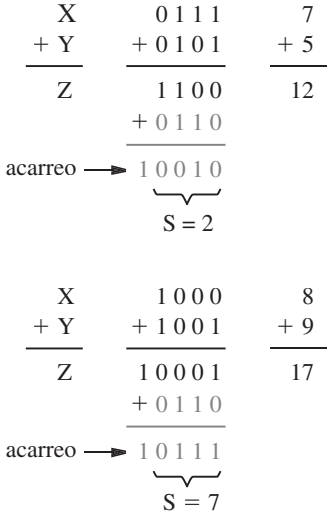


Figura 5.35 Suma de dígitos BCD.

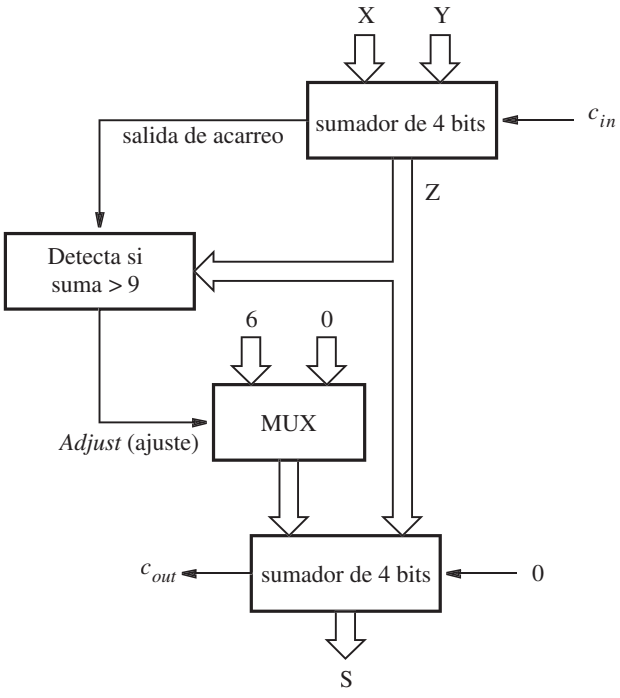


Figura 5.36 Diagrama de bloques para un sumador BCD de un dígito.

En la figura 5.37 se muestra una implementación de este diagrama de bloques usando código de VHDL. Las entradas *X* y *Y* se definen como números de cuatro bits. La salida suma, *S*, se define como un número de cinco bits, lo que permite que la salida de acarreo aparezca en el bit S_4, en tanto que la suma se produce en los bits S_{3-0}. La suma intermedia *Z* también se define como un número de cinco bits. Recuérdese de lo expuesto en la sección 5.5.4 que VHDL requiere que al menos uno de los operandos de una operación aritmética tenga el mismo número de bits que el resultado. Este requisito explica por qué es preciso concatenar un 0 a la entrada X en la expresión Z <= ('0' & X) + Y.

La instrucción

```
<= '1' WHEN Z > 9 ELSE '0'
```

usa un tipo de instrucción de asignación de señal de VHDL que no hemos visto. Se llama *asignación de señal seleccionada* y se usa para asignar uno de múltiples valores a una señal, con base en cierto criterio. En este caso el criterio es la condición $Z > 9$. Si esta condición se satisface, la instrucción asigna 1 a *Adjust*; de otro modo, le asigna 0. Otros ejemplos de la asignación de señal seleccionada se brindan en el capítulo 6.

También cabe notar que incluimos la señal *Adjust* en el código de VHDL sólo para que fuera consistente con la figura 5.36. Pudimos haberla eliminado fácilmente y escribir la expresión como

```
S <= Z WHEN Z < 10 ELSE Z + 6
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCD IS
    PORT ( X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(4 DOWNTO 0) ) ;
END BCD ;

ARCHITECTURE Behavior OF BCD IS
    SIGNAL Z : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL Adjust : STD_LOGIC ;
BEGIN
    Z <= ('0' & X) + Y ;
    Adjust <= '1' WHEN Z > 9 ELSE '0' ;
    S <= Z WHEN (Adjust = '0') ELSE Z + 6 ;
END Behavior
```

Figura 5.37 Código de VHDL para un sumador BCD de un dígito.

Para comprobar la exactitud funcional del código se realiza una simulación funcional. En la figura 5.38 se ofrece un ejemplo de los resultados obtenidos.

Si se desea derivar un circuito para implementar el diagrama de bloques de la figura 5.36 a mano, en vez de usar VHDL, entonces podemos emplear el enfoque siguiente. Para definir la función *Adjust*, obsérvese que la suma intermedia superará 9 si la salida de acarreo proveniente del sumador de cuatro bits es igual a 1, o si $z_3 = 1$ y z_2 o z_1 (o ambos) son iguales a 1. Por tanto, la expresión lógica para esta función es

$$Adjust = \text{salida de carreo} + z_3(z_2 + z_1)$$

En lugar de implementar otro sumador completo de cuatro bits para realizar la corrección, podemos usar un circuito más simple porque la suma de una constante 6 no requiere la capacidad completa de un sumador de cuatro bits. Nótese que el bit menos significativo de la suma, s_0, no se afecta en absoluto; por tanto, $s_0 = z_0$. Se puede usar un sumador de dos bits para desarrollar los bits s_2 y s_1. El bit s_3 es el mismo que z_3 si la salida de acarreo proveniente del sumador de dos bits es 0, y es igual a $\overline{z}_3$ si esta salida de acarreo es igual a 1. En la figura 5.39 se muestra

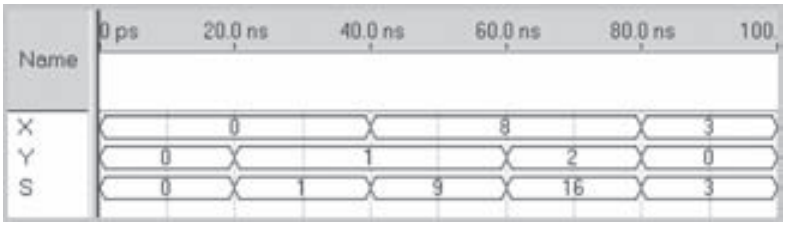


Figura 5.38 Simulación funcional del código de VHDL de la figura 5.37.

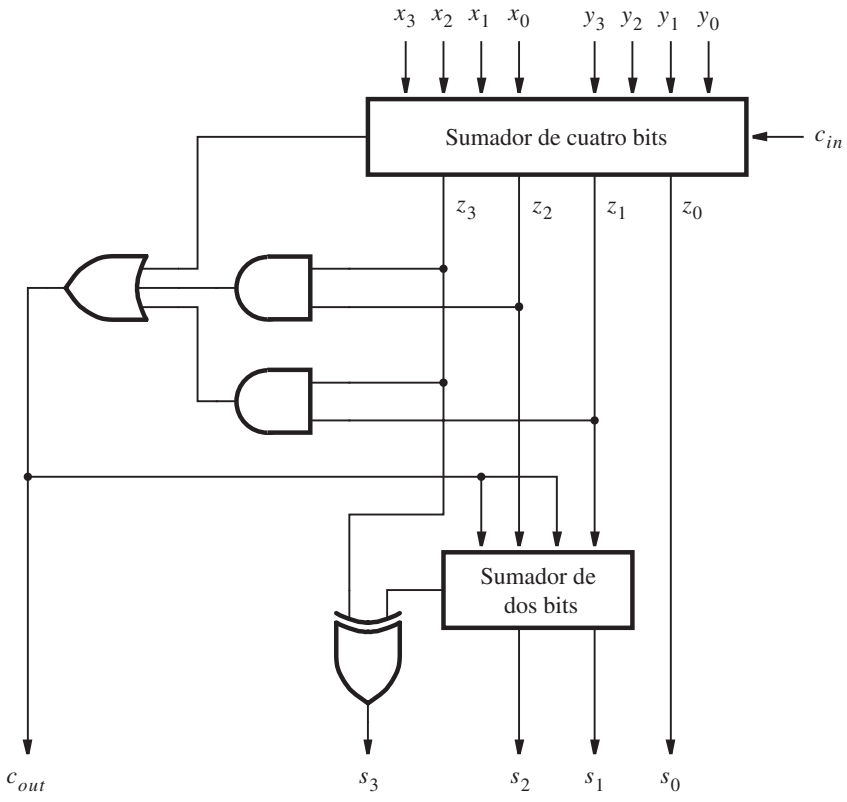


Figura 5.39 Circuito para un sumador BCD de un dígito.

un circuito completo que implementa este esquema. Al usar el sumador BCD de un dígito como bloque básico es posible construir sumadores BCD más grandes de la misma forma en que se utiliza un sumador completo binario para construir sumadores binarios con acarreo en cascada más grandes.

La resta de números BCD puede manejarse con el enfoque de complemento a la base. Igual que empleamos la representación en complemento a 2 para manejar los números binarios negativos, podemos usar la representación en complemento a 10 para manejar los decimales. El desarrollo de tal esquema se deja como ejercicio al lector (véase el problema 5.19).

5.8 CÓDIGO DE CARACTERES ASCII

El código más popular para representar información en sistemas digitales se usa para letras y para números, así como para algunos caracteres de control. Se conoce como *código ASCII*, que significa American Standard Code for Information Interchange (Código Americano Estándar para el Intercambio de Información). En la tabla 5.4 se presenta el código especificado por este estándar.

Tabla 5.4 Código ASCII de siete bits.

Posiciones de bit	Posiciones de bit 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	ESPACIO	0	@	P	′	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	~
1111	SI	US	/	?	O	—	°	DEL

NUL	Nulo/Espera	SI	Salir corrimiento
SOH	Comienzo de encabezado	DLE	Ingresar corrimiento
STX	Comienzo de texto	DC1-DC4	Escape de liga de datos
ETX	Fin de texto	NAK	Control de dispositivo
EOT	Fin de transmisión	SYN	Confirmación negativa
ENQ	Petición	ETB	Fin de bloque transmitido Espera síncrona
ACQ	Confirmación	CAN	Cancelar (error en datos)
BEL	Señal audible	EM	Fin de medio
BS	Retroceso	SUB	Secuencia especial
HT	Horizontal tab	ESC	Escape
LF	Tabulador horizontal	FS	Separador de archivo
VT	Salto de línea	GS	Separador de grupo
FF	Tabulador vertical	RS	Separador de registro
CR	Avance de página	US	Separador de unidad
SO	Retorno de carro	DEL	Borrar/Espera

Posiciones de bit de formato de código = 6 5 4 3 2 1 0

El código ASCII usa patrones de siete bits para denotar 128 caracteres. Diez de ellos son dígitos decimales de 0 a 9. Nótese que los bits de orden superior tienen el mismo patrón, $b_6b_5b_4 = 011$, para los 10 dígitos. Cada dígito se identifica mediante los cuatro bits de orden inferior, b_{3-0}, que usan los patrones binarios para esos dígitos. Las letras mayúsculas y minúsculas se codifican de una manera que facilita el ordenamiento textual de la información. Los códigos para la A a la Z están en secuencia numérica ascendente, lo que significa que la tarea de ordenar letras (o palabras) se logra mediante una simple comparación aritmética de los códigos que las representan.

Los caracteres que son letras del alfabeto o números se conocen como caracteres *alfanuméricos*. Además de ellos, en el código ASCII se incluyen signos de puntuación como ! y ?; signos que se usan comúnmente, como & y %; y un juego de caracteres de control. Los caracteres de control son los que se necesitan en los sistemas de cómputo para manejar y transferir datos entre dispositivos. Por ejemplo, el carácter de retorno de carro, que se abrevia CR en la tabla, indica que el carro, o posición del cursor, de un dispositivo de salida, digamos una impresora o un monitor, debe regresar a la columna del extremo izquierdo.

El código ASCII se usa para codificar información que se maneja como texto. No es práctico para la representación de números que se emplean como operandos en operaciones aritméticas. Para este propósito es mejor convertir los números codificados en ASCII en una representación binaria, como explicamos antes.

El estándar ASCII utiliza siete bits para codificar un carácter. En los sistemas de cómputo, un tamaño más natural es de ocho bits, o un byte. Hay dos formas comunes de encajar un carácter codificado en ASCII en un byte. Una es establecer el octavo bit, b_7, en 0. Otra consiste en usar ese bit para indicar la paridad de los otros siete bits, lo que significa mostrar si el número de 1 en el código de siete bits es par o impar.

Paridad

El concepto de *paridad* se usa mucho en los sistemas digitales con propósitos de comprobación de errores. Cuando la información digital se transmite de un punto a otro, quizá a través de largos cables, es posible que algunos bits se corrompan durante la transmisión. Por ejemplo, el remitente puede transmitir un bit cuyo valor es igual a 1, pero el receptor observa un bit cuyo valor es 0. Supóngase que un elemento de datos consta de n bits. Es posible implementar un simple mecanismo de comprobación de errores incluyendo un bit adicional, p, que indica la paridad del elemento de n bits. Se pueden usar dos tipos de paridad. Para *paridad par*, se da al bit p el valor tal que el número total de 1 en los $n + 1$ bits transmitidos (que comprenden los datos de n bits y el bit de paridad p) sea par. Para *paridad impar*, se da al bit p el valor que hace que el número total de 1 sea impar. El remitente genera el bit p con base en el elemento de datos de n bits que se transmitirá. El receptor comprueba si la paridad del elemento recibido es correcta.

Los circuitos que generan y comprueban la paridad pueden realizarse con compuertas XOR. Por ejemplo, para un elemento de datos de cuatro bits que consta de los bits $x_3x_2x_1x_0$, el bit de paridad par puede generarse como

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

En el extremo receptor, la comprobación se realiza con

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

Si $c = 0$, entonces el elemento recibido muestra la paridad correcta. Si $c = 1$, entonces ocurrió un error. Nótese que observar $c = 0$ no es garantía de que el elemento recibido sea correcto. Si

dos o cualesquier número par de bits invierten sus valores durante la transmisión, la paridad del elemento de datos no cambiará; por tanto, el error no será detectado. Pero si se corrompe un número impar de bits, entonces sí se detectará.

El atractivo de la comprobación de paridad radica en su simpleza. Existen otros esquemas más refinados que proporcionan mecanismos de comprobación de errores más confiables [4]. En la sección 9.3 veremos de nuevo los circuitos de paridad.

5.9 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas usuales que el lector puede encontrar, y se muestra cómo resolverlos.

Ejemplo 5.7 **Problema:** Convierta el número decimal 14959 en número hexadecimal.

Solución: Un entero se convierte en representación hexadecimal mediante divisiones sucesivas entre 16, de modo que en cada paso el residuo sea un dígito hexa. Para ver por qué es cierto esto, considere un número de cuatro dígitos $H = h_3h_2h_1h_0$. Su valor es

$$V = h_3 \times 16^3 + h_2 \times 16^2 + h_1 \times 16 + h_0$$

Si dividimos esto entre 16 se obtiene

$$\frac{V}{16} = h_3 \times 16^2 + h_2 \times 16 + h_1 + \frac{h_0}{16}$$

Por tanto, el residuo da h_0. En la figura 5.40 se muestran los pasos necesarios para realizar la conversión $(14959)_{10} = (3A6F)_{16}$.

Convertir $(14959)_{10}$

			Residuo	Dígito hexa	
14959 ÷ 16	=	934	15	F	LSB
934 ÷ 16	=	58	6	6	
58 ÷ 16	=	3	10	A	
3 ÷ 16	=	0	3	3	MSB

El resultado es $(3A6F)_{16}$

Figura 5.40 Conversión de decimal en hexadecimal.

Ejemplo 5.8

Problema: Convierta la fracción decimal 0.8254 en representación binaria.
Solución: Como se indicó en la sección 5.7.1, una fracción binaria se representa como el patrón de bit $B = 0.b_{-1}b_{-2}\cdots b_{-m}$ y su valor es

$$V = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

Al multiplicar esta expresión por 2 se produce

$$b_{-1} + b_{-2} \times 2^{-1} + \cdots + b_{-m} \times 2^{-(m-1)}$$

Aquí, el término del extremo izquierdo es el primer bit a la derecha del punto de la base. Los términos restantes constituyen otra fracción binaria que puede manipularse de la misma forma. Por tanto, para convertir una fracción decimal en binaria se multiplica el número decimal por 2 y el bit calculado se establece en 0 si el producto es menor que 1, y se pone en 1 si el producto es mayor o igual a 1. Este cálculo se repite hasta obtener un número suficiente de bits para satisfacer la precisión deseada. Note que puede no ser posible representar una fracción decimal con una fracción binaria que tenga exactamente el mismo valor. En la figura 5.41 se muestra el cálculo requerido que produce $(0.8254)_{10} = (0.11010011\ldots)_2$.

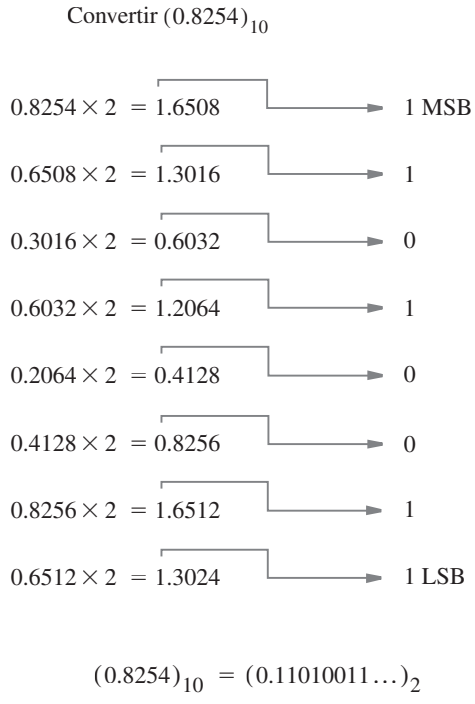


Figura 5.41 Conversión de fracciones de decimal a binario.

Ejemplo 5.9 **Problema:** Convierta el número decimal con punto fijo 214.45 en un número binario con punto fijo.

Solución: Para la parte entera se realizan divisiones sucesivas entre 2, como se ilustra en la figura 5.1. Para la parte fraccionaria se efectúan multiplicaciones sucesivas por 2, como se describe en el ejemplo 5.8. El cálculo completo se presenta en la figura 5.42, lo que produce $(214.45)_{10} = (11010110.0111001\ldots)_2$.

Ejemplo 5.10 **Problema:** En los cálculos en computadora con frecuencia es necesario comparar números. Dos números con signo de cuatro bits, $X = x_3x_2x_1x_0$ y $Y = y_3y_2y_1y_0$, pueden compararse mediante el circuito restador de la figura 5.43, que realiza la operación $X - Y$. Las tres salidas denotan lo siguiente:

- $Z = 1$ si el resultado es 0; de otro modo $Z = 0$
- $N = 1$ si el resultado es negativo; de otro modo $N = 0$
- $V = 1$ si ocurre desbordamiento aritmético; de otro modo $V = 0$

Muestre cómo pueden usarse Z, N y V para determinar los casos $X = Y, X < Y, X \leq Y, X > Y$, y $X \geq Y$.

Solución: Considere primero el caso $X <$ Y, donde pueden surgir las posibilidades siguientes:

- Si X y Y tienen el mismo signo no habrá desbordamiento; por tanto, $V = 0$. Entonces, para X y Y positivos y negativos la diferencia será negativa ($N = 1$).
- Si X es negativa y Y positiva, la diferencia será negativa ($N = 1$) si no hay desbordamiento ($V = 0$); pero el resultado será positivo ($N = 0$) si hay desbordamiento ($V = 1$).

En consecuencia, si $X < Y$, entonces $N \oplus V = 1$.

El caso $X = Y$ se detecta con $Z = 1$. Entonces, $X \leq Y$ se detecta mediante $Z + (N \oplus V) = 1$. Los dos últimos casos sólo son simples inversos: $X > Y$ si $Z + (N \oplus V) = 1$ y $X \geq Y$ si $\overline{N \oplus V} = 1$.

Ejemplo 5.11 **Problema:** Escriba el código de VHDL para especificar el circuito de la figura 5.43.

Solución: Es posible especificar el circuito mediante el enfoque estructural presentado en la figura 5.26, como se indica en la figura 5.44. Los cuatro sumadores completos se definen en un paquete en la figura 5.24.

Este enfoque se complica cuando se incluyen grandes circuitos, como sería el caso si el comparador tuviese operandos de 32 bits. Una posibilidad consiste en usar una especificación por comportamiento, como se muestra en la figura 5.45, que se basa en el esquema de la figura 5.28. Note que hemos establecido directamente que Y debe restarse de X, de modo que no tenemos que complementar Y de manera explícita. Como el compilador de VHDL implementará el circuito usando un módulo de biblioteca, es preciso especificar la señal desbordamiento, V, sólo en términos de los bits S, ya que las señales de acarreo entre etapas no son accesibles como se explicó para la figura 5.28.

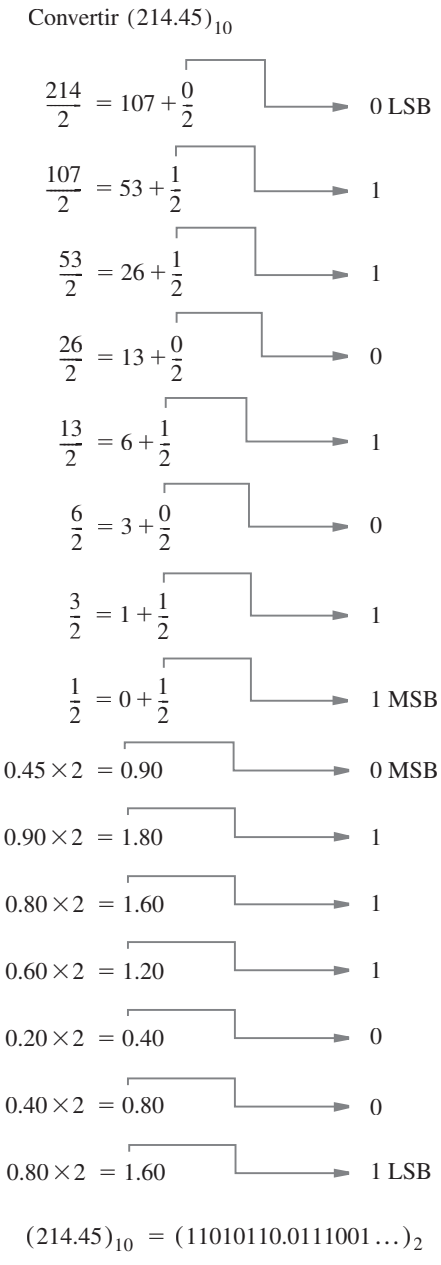


Figura 5.42 Conversión de números de punto fijo de decimal a binario.

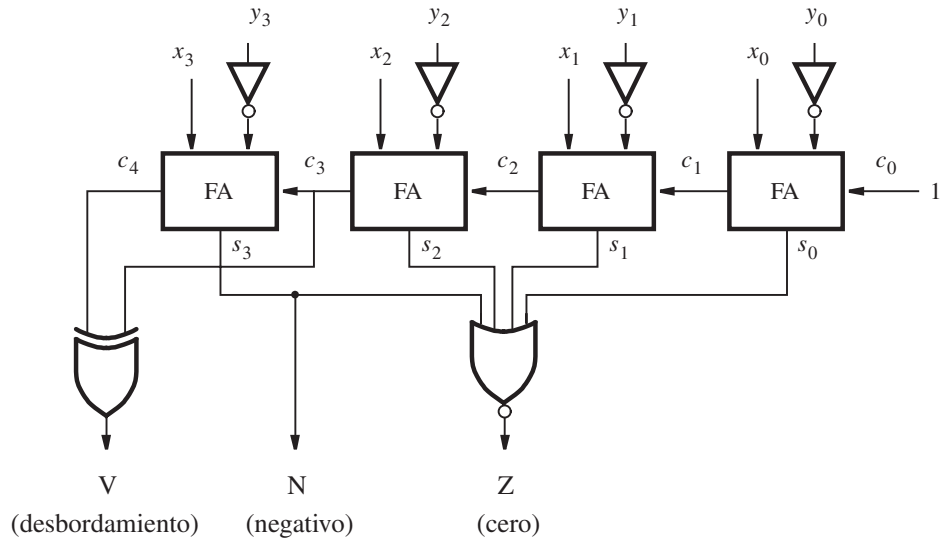


Figura 5.43 Circuito comparador.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY comparator IS
    PORT ( X, Y     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z  : OUT   STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Structure OF comparator IS
    SIGNAL S : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 4) ;
BEGIN
    stage0: fulladd PORT MAP ( '1', X(0), NOT Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), NOT Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), NOT Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), NOT Y(3), S(3), C(4) ) ;
    V <= C(4) XOR C(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = "0000" ELSE '0';
END Structure ;
```

Figura 5.44 Código estructural de VHDL para el circuito comparador.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY comparator IS
    PORT ( X, Y     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z  : OUT   STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Behavior OF comparator IS
    SIGNAL S : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    S <= ('0' & X) + Y ;
    V <= S(4) XOR X(3) XOR Y(3) XOR S(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = 0 ELSE '0';
END Behavior ;
```

Figura 5.45 Código VHDL por comportamiento para el circuito comparador.

Ejemplo 5.12

Problema: En la figura 5.32 se muestra un circuito multiplicador de cuatro bits. Cada fila consta de cuatro bloques sumadores completos (FA) conectados en una configuración de acarreo en cascada. El retraso ocasionado por las señales de acarreo que caen en cascada a través de las filas tiene un efecto significativo en el tiempo necesario para generar el producto salida. Con la intención de agilizar el circuito, es posible usar el ordenamiento de la figura 5.46. Aquí, los acarreos en una fila se "guardan" e incluyen en la fila siguiente en la posición de bit correcta. Luego, en la primera fila los sumadores completos pueden usarse para sumar tres bits corridos adecuadamente del multiplicando según los seleccionen los bits multiplicadores. Por ejemplo, en la posición de bit 2 las tres entradas son m_2q_0, m_1q_1 y m_0q_2. En la última fila aún es necesario usar el sumador con acarreo en cascada. Un circuito que consta de un arreglo de sumadores completos conectados de esta forma se llama *arreglo sumador de acarreo guardado*.

¿Cuál es el retraso total del circuito en la figura 5.46, comparado con el del circuito de la figura 5.32?

Solución: En el circuito de la figura 5.32a la trayectoria más larga es a través de los dos sumadores completos del extremo derecho de la fila superior, seguida por los dos FA más a la derecha en la segunda fila, y luego por los cuatro FA de la fila inferior. Por tanto, este retraso es ocho veces el retraso a través de un bloque sumador completo. Además, hay un retraso de compuerta AND necesario para formar las entradas al primer FA en la fila superior. Estos retrasos combinados son el retraso crítico, que determina la velocidad del circuito multiplicador.

En el circuito de la figura 5.46, la trayectoria más larga es a través de los FA más a la derecha de la primera y segunda filas, seguida por los cuatro FA de la fila inferior. Por tanto, el retraso crítico es seis veces el retraso a través de un bloque sumador completo más el retraso de la compuerta AND necesario para formar las entradas al primer FA de la fila superior.

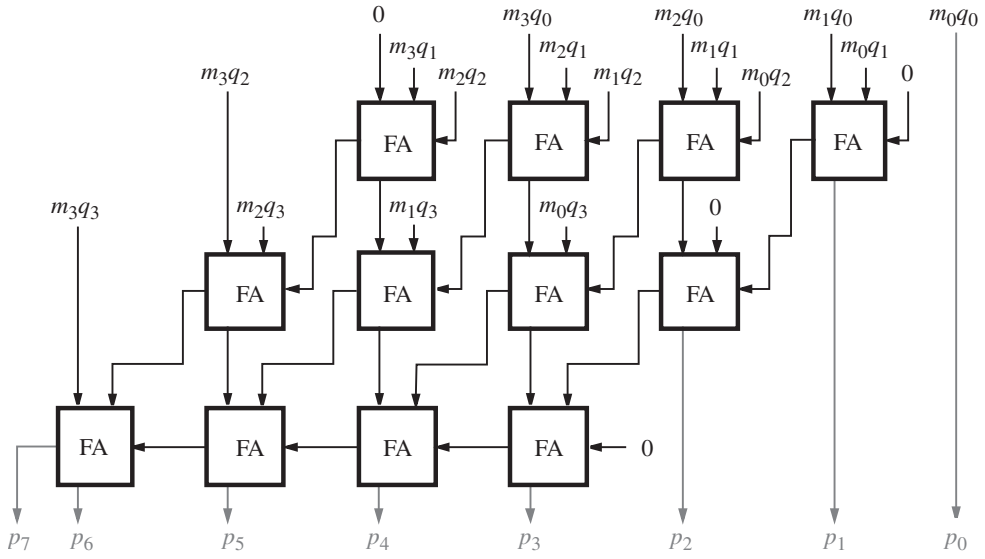


Figura 5.46 Arreglo multiplicador de acarreo guardado.

PROBLEMAS

Al final del libro se proporcionan las respuestas a los problemas marcados con asterisco.

***5.1** Determine los valores decimales de los siguientes números sin signo:

a) $(0111011110)_2$
b) $(1011100111)_2$
c) $(3751)_8$
d) $(A25F)_{16}$
e) $(F0F0)_{16}$

***5.2** Determine los valores decimales de los siguientes números en complemento a 1:

a) 0111011110
b) 1011100111
c) 1111111110

***5.3** Determine los valores decimales de los siguientes números en complemento a 2:

a) 0111011110
b) 1011100111
c) 1111111110

***5.4** Convierta los números decimales 73, 1906, −95 y −1630 en números con signo de 12 bits en las representaciones siguientes:

a) Signo y magnitud
b) Complemento a 1
c) Complemento a 2

5.5 Realice las operaciones siguientes que implican números en complemento a 2 de ocho bits e indique si ocurre desbordamiento aritmético. Compruebe sus respuestas convirtiendo en representación decimal signo y magnitud.

00110110	01110101	11011111
+ 01000101	+ 11011110	+ 10111000
00110110	01110101	11010011
− 00101011	− 11010110	− 11101100

5.6 Pruebe que la operación XOR es asociativa, lo que significa que $x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$.

5.7 Demuestre que el circuito de la figura 5.5 implementa el sumador completo especificado en la figura 5.4*a*.

5.8 Pruebe la validez de la simple regla para encontrar el complemento a 2 de un número presentada en la sección 5.3. Recuerde que la regla establece que al revisar un número de derecha a izquierda, todos los 0 y el primer 1 se copian; luego todos los bits restantes se complementan.

5.9 Pruebe la validez de la expresión Desbordamiento (*overflow*) $= c_n \oplus c_{n-1}$ para la suma de números con signo de n bits.

5.10 En la sección 5.5.4 establecimos que una señal de acarreo, c_k, a partir de la posición de bit $k - 1$ de un circuito sumador puede generarse como $c_k = x_k \oplus y_k \oplus s_k$, donde x_k y y_k son entradas y s_k es el bit suma. Compruebe la exactitud de esta afirmación.

***5.11** Considere el circuito de la figura P5.1. ¿Puede usarse este circuito como una etapa en un sumador con acarreo en cascada? Discuta las ventajas y las desventajas.

***5.12** Determine el número de compuertas necesarias para implementar un sumador con acarreo de adelanto de n bits, si no se suponen restricciones de carga de entrada. Use compuertas AND, OR y XOR con cualquier número de entradas.

***5.13** Determine el número de compuertas necesarias para implementar un sumador con acarreo de adelanto de ocho bits, si se supone que la máxima entrada de carga para las compuertas es cuatro.

5.14 En al figura 5.18 se presentó la estructura de un sumador jerárquico con acarreo de adelanto. Muestre el circuito completo para una versión de cuatro bits de este sumador, construido con dos bloques de dos bits.

5.15 ¿Cuál es la trayectoria de retraso crítico en el multiplicador de la figura 5.32? ¿Cuál es el retraso a lo largo de esta trayectoria en términos del número de compuertas?

5.16 *a*) Escriba una entidad de VHDL para describir el bloque de circuito de la figura 5.32*b*. Use las herramientas CAD para sintetizar un circuito a partir del código y compruebe su exactitud funcional.

b) Escriba una entidad de VHDL para describir el bloque de circuito de la figura 5.32*c*. Use las herramientas CAD para sintetizar un circuito a partir del código y compruebe su exactitud funcional.

c) Escriba una entidad de VHDL para describir el multiplicador de 4×4 de la figura 5.32*a*. Su código debe ser jerárquico y usar los subíndices diseñados en los incisos *a*) y *b*). Sintetice un circuito a partir del código y compruebe su exactitud funcional.

***5.17** Considere el código de VHDL de la figura P5.2. Dada la relación entre las señales IN y OUT, ¿cuál es la funcionalidad del circuito descrito por el código? Comente si este código constituye o no un buen estilo para la funcionalidad que representa.

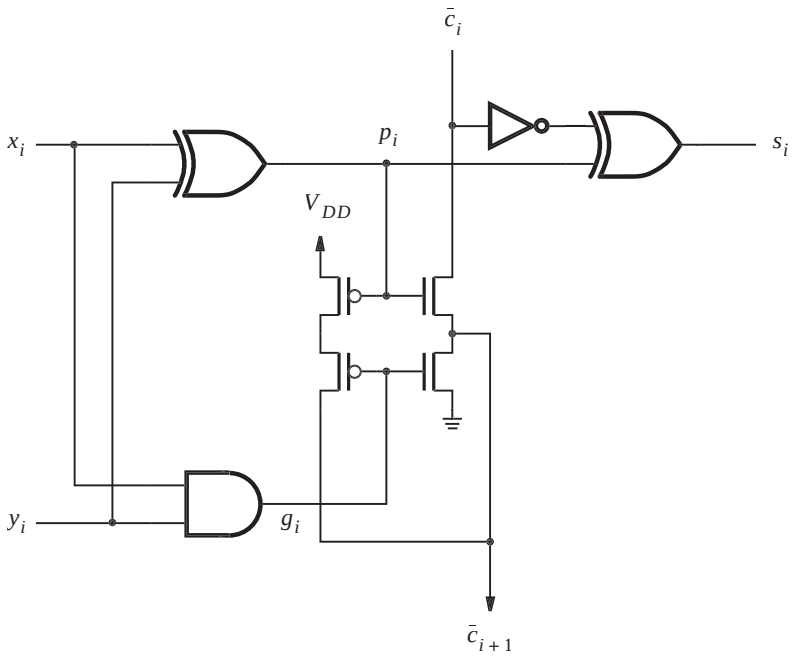


Figura P5.1 Circuito para el problema 5.11.

5.18 Diseñe un circuito que genere el complemento a 9 de un dígito BCD. Note que el complemento a 9 de d es $9 - d$.

5.19 Derive un esquema para realizar la resta usando operandos BCD. Muestre un diagrama de bloques para el circuito restador.

Sugerencia: La resta puede realizarse fácilmente si los operandos están en representación de complemento a 10 (complemento a la base). En esta representación, el dígito signo es 0 para un número positivo y 9 para un número negativo.

5.20 Escriba código completo de VHDL para el circuito que derivó en el problema 5.19.

***5.21** Suponga que se quiere determinar cuántos bits en un número sin signo de tres bits son iguales a 1. Diseñe el circuito más simple que pueda realizar esa tarea.

5.22 Repita el problema 5.21 para un número sin signo de seis bits.

5.23 Repita el problema 5.21 para un número sin signo de ocho bits.

5.24 Muestre una interpretación gráfica de números decimales de tres dígitos, similar a la figura 5.12. El dígito más a la izquierda es 0 para números positivos y 9 para negativos. Compruebe la validez de la respuesta probando algunos ejemplos de suma y resta.

5.25 El sistema numérico ternario tiene tres dígitos: 0, 1, y 2. En la figura P5.3 se define un medio sumador ternario. Diseñe un circuito que implemente este medio sumador usando señales codificadas en binario, tal que dos bits se usen para cada dígito ternario. Sean $A = a_1a_0$, $B = b_1b_0$ y $Sum = s_1s_0$; note que *Carry* es sólo una señal binaria. Use la codificación siguiente: $00 = (0)_3$, $01 = (1)_3$ y $10 = (2)_3$. Minimice el costo del circuito.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( Input  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Output : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END problem ;

ARCHITECTURE LogicFunc OF problem IS
BEGIN
    WITH Input SELECT
        Output<= "0001" WHEN "0101",
                 "0010" WHEN "0110",
                 "0011" WHEN "0111",
                 "0010" WHEN "1001",
                 "0100" WHEN "1010",
                 "0110" WHEN "1011",
                 "0011" WHEN "1101",
                 "0110" WHEN "1110",
                 "1001" WHEN "1111",
                 "0000" WHEN OTHERS ;
END LogicFunc ;
```

Figura P5.2 Código para el problema 5.17.

A B	*Carry*	*Sum*
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figura P5.3 Medio sumador ternario.

5.26 Diseñe un circuito sumador completo ternario aplicando el enfoque descrito en el problema 5.25.

5.27 Considere las restas 26 – 27 = 99 y 18 – 34 = 84. Con los conceptos presentados en la sección 5.3.4, explique cómo pueden interpretarse estas respuestas (99 y 84) como los resultados con signos correctos de tales restas.

BIBLIOGRAFÍA

1. V. C. Hamacher, Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 5a. ed. (McGraw-Hill: Nueva York, 2002).
2. D. A. Patterson y J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2a. ed. (Morgan Kaufmann: San Francisco, CA, 1998).
3. Intitute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic", *Computer* 14, núm. 3 (marzo de 1981), pp. 51-62.
4. W. W. Peterson y E. J. Weldon Jr., *Error-Correcting Codes*, 2a. ed. (MIT Press: Boston, MA, 1972).

capítulo

6

BLOQUES CONSTRUCTORES DE CIRCUITOS COMBINACIONALES

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

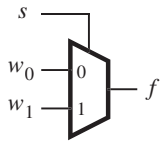
- Los subcircuitos combinacionales comúnmente usados
- Los multiplexores, que pueden usarse para la selección de señales y la implementación de funciones lógicas generales
- Los circuitos empleados para codificar, decodificar y convertir código
- Los constructores principales de VHDL que se utilizan para definir circuitos combinacionales

En los capítulos anteriores expusimos las técnicas básicas para diseñar circuitos lógicos. En la práctica, pocos tipos de circuitos lógicos se usan con frecuencia como bloques constructores en diseños más grandes. En este capítulo se exponen estos bloques y se brindan ejemplos de su empleo. Asimismo, se incluye una gran sección acerca de VHDL, que describe varias características principales del lenguaje.

6.1 MULTIPLEXORES

En los capítulos 2 y 3 presentamos brevemente los multiplexores. Un circuito multiplexor tiene varias entradas de datos, una o más entradas de selección y una salida. Pasa el valor de señal de una de las entradas de datos a la salida. La entrada de datos se elige mediante los valores de las entradas de selección. En la figura 6.1 se muestra un multiplexor dos a uno. En el inciso *a*) se presenta el símbolo que suele usarse para representar un multiplexor. La entrada *select* (selección), s, elige como salida del multiplexor cualquiera de las entradas w_0 o w_1. La funcionalidad del multiplexor queda descrita con una tabla de verdad, como se muestra en el inciso *b*) de la figura. En el inciso *c*) se presenta una implementación en suma de productos del multiplexor dos a uno, y en el inciso *d*) se ilustra cómo construirlo con compuertas de transmisión.

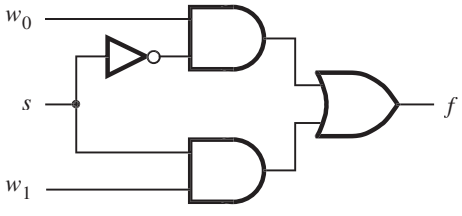
En la figura 6.2*a* se describe un multiplexor más grande con cuatro entradas de datos, $w_0, \ldots, w_3$, y dos entradas de selección, s_1 y s_0. Como se muestra en la tabla de verdad del inciso *b*) de la figura, los números de dos bits representados por $s_1 s_0$ escogen una de las entradas de



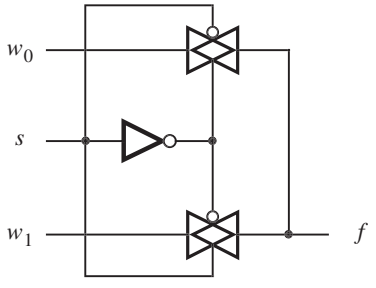
a) Símbolo gráfico

s	f
0	w_0
1	w_1

b) Tabla de verdad

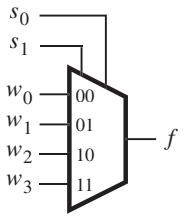


c) Circuito en suma de productos



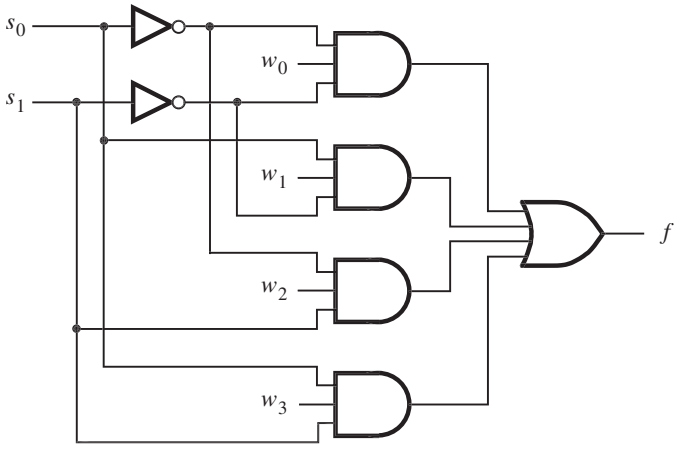
d) Circuito con compuertas de transmisión

Figura 6.1 Multiplexor dos a uno.



s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

a) Símbolo gráfico *b*) Tabla de verdad



c) Circuito

Figura 6.2 Multiplexor cuatro a uno.

datos como la salida del multiplexor. En la figura 6.2*c* aparece una implementación en suma de productos de un multiplexor cuatro a uno. Esa implementación realiza la función multiplexora

$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

Es posible construir multiplexores más grandes con el mismo enfoque. En general, el número de entradas de datos, n, es una potencia entera de dos. Un multiplexor con n entradas de datos, $w_0, \ldots, w_{n-1}$, requiere $\lceil \log_2 n \rceil$ entradas de selección. Los multiplexores más grandes también pueden construirse a partir de otros más pequeños. Por ejemplo, el multiplexor cuatro a uno puede construirse con tres multiplexores dos a uno, como se ilustra en la figura 6.3. Si el multiplexor cuatro a uno se implementa con compuertas de transmisión, entonces la estructura de esta figura siempre se utiliza. En la figura 6.4 se muestra cómo se construye un multiplexor 16 a 1 con cinco multiplexores cuatro a uno.

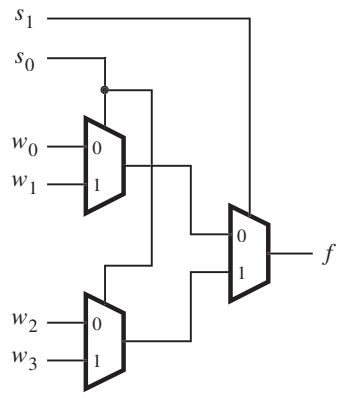


Figura 6.3 Uso de multiplexores dos a uno para construir un multiplexor cuatro a uno.

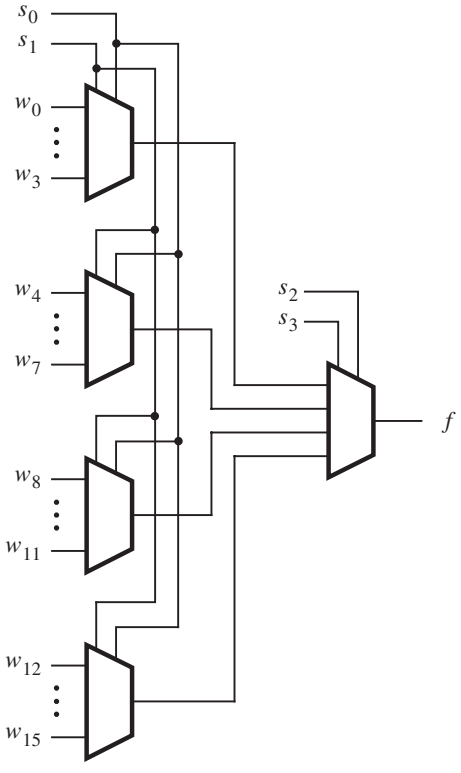


Figura 6.4 Multiplexor 16 a 1.

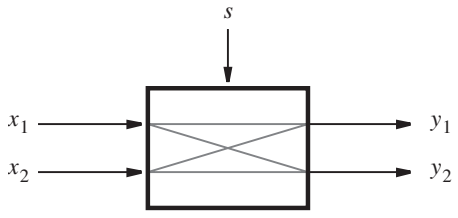
Ejemplo 6.1

En la figura 6.5 se muestra un circuito con dos entradas, x_1 y x_2, y dos salidas, y_1 y y_2. Como indican las líneas grises, la función del circuito es permitir que alguna de las entradas se conecte a alguna de las salidas, bajo el control de otra entrada, s. Un circuito con n entradas y k salidas, cuya única función sea proporcionar una capacidad de conectar cualquier entrada a cualquier salida, se conoce como interruptor cruzado $n \times k$. Es posible crear interruptores cruzados de varios tamaños, con diferente número de entradas y salidas. Cuando hay dos entradas y dos salidas se llama interruptor cruzado 2×2.

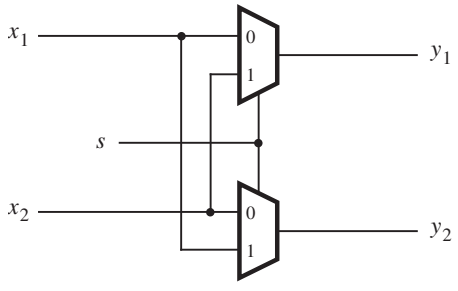
En la figura 6.5*b* se indica cómo implementar el interruptor cruzado 2×2 con multiplexores dos a uno. Las entradas de selección del multiplexor se controlan mediante la señal s. Si $s = 0$, el interruptor cruzado conecta x_1 a y_1 y x_2 a y_2, mientras que si $s = 1$, el interruptor cruzado conecta x_1 a y_2 y x_2 a y_1. Los interruptores cruzados son útiles en muchas aplicaciones prácticas en las que es necesario tener capacidad de conectar un juego de cables a otro, donde el patrón de conexión cambia de tiempo en tiempo.

Ejemplo 6.2

En la sección 3.6.5 expusimos los chips de arreglo de compuertas programables por campo (FPGA). En la figura 3.39 se observa un pequeño FPGA programado para implementar un circuito específico. Los bloques lógicos del FPGA tienen dos entradas y hay cuatro pistas en cada canal de enrutamiento. Cada uno de los interruptores programables que conecta una entrada o salida de bloque lógico a un cable de interconexión se indica con una X. Una pequeña parte de la figura 3.39 se reproduce en la figura 6.6*a*. Por claridad, en la figura únicamente se muestra un

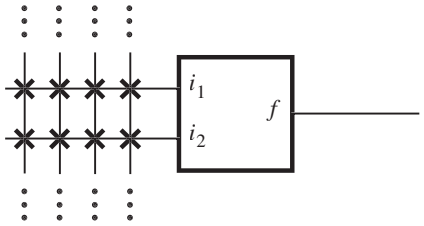


a) Interruptor cruzado 2×2

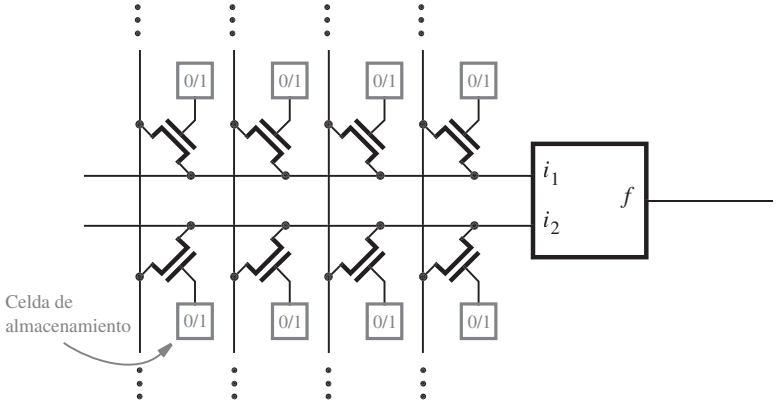


b) Implementación con multiplexores

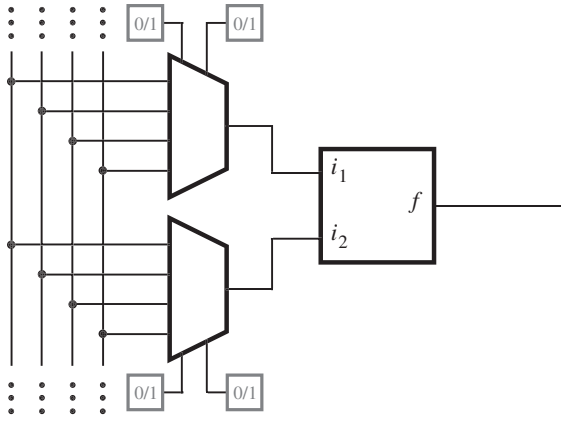
Figura 6.5 Aplicación práctica de multiplexores.



a) Parte del FPGA de la figura 3.39



b) Implementación con transistores de paso



c) Implementación con multiplexores

Figura 6.6 Implementación de interruptores programables en un FPGA.

solo bloque lógico y los cables de interconexión e interruptores asociados con sus terminales de entrada.

En la figura 6.6*b* se ilustra una forma en la que los interruptores programables pueden implementarse. Cada X en el inciso *a*) de la figura se realiza con un transistor NMOS controlado por una celda de almacenamiento. Este tipo de interruptor programable también se presentó en la figura 3.68. En la sección 3.6.5 describimos brevemente las celdas de almacenamiento, y las explicaremos con más detalle en la sección 10.1. Cada celda almacena un solo valor lógico, 0 o 1, y lo ofrece como la salida de la celda. Cada celda de almacenamiento se construye con varios transistores. Por ende, las ocho celdas mostradas en la figura usan una cantidad significativa del área del chip.

El número de celdas de almacenamiento necesarias puede reducirse si se emplean multiplexores, como se muestra en la figura 6.6*c*. Cada entrada de bloque lógico se alimenta mediante el multiplexor cuatro a uno, y las entradas de selección se controlan por medio de celdas de almacenamiento. Este enfoque requiere sólo cuatro celdas de almacenamiento, en lugar de ocho. En los FPGA comerciales casi siempre se adopta el enfoque basado en multiplexores.

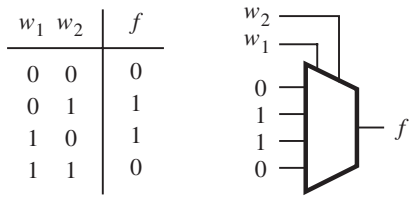
6.1.1 SÍNTESIS DE FUNCIONES LÓGICAS MEDIANTE MULTIPLEXORES

Los multiplexores son útiles en muchas aplicaciones prácticas, como las descritas anteriormente. También pueden usarse en una forma más general para sintetizar funciones lógicas. Considérese el ejemplo de la figura 6.7*a*. La tabla de verdad define la función $f = w_1 \oplus w_2$. que puede implementarse mediante un multiplexor cuatro a uno en el que los valores de f de cada fila de la tabla de verdad se conectan como constantes a las entradas de datos del multiplexor. Las entradas de selección del multiplexor son manejadas por w_1 y w_2. Por tanto, para cada combinación de w_1w_2, la salida f es igual al valor de la función en la fila correspondiente de la tabla de verdad.

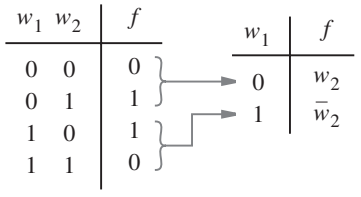
La implementación anterior es directa, mas no muy eficiente. Una mejor puede derivarse manipulando la tabla de verdad como se indica en la figura 6.7*b*, que permite que f se implemente mediante un solo multiplexor dos a uno. Una de las señales de entrada, w_1 en este ejemplo, se elige como la entrada de selección del multiplexor dos a uno. La tabla de verdad se dibuja de nuevo para indicar el valor de f para cada valor de w_1. Cuando $w_1 = 0$, f tiene el mismo valor que la entrada w_2, y cuando $w_1 = 1$, f tiene el valor de $\overline{w}_2$. En la figura 6.7*c* se presenta el circuito que implementa esta tabla. Este procedimiento puede aplicarse para sintetizar un circuito que implemente cualquier función lógica.

Ejemplo 6.3

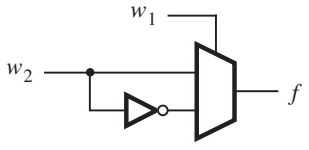
En la figura 6.8*a* se presenta la tabla de verdad de la función original de tres entradas, y se muestra cómo modificar la tabla de verdad para implementarla con un multiplexor cuatro a uno. Es posible elegir cualesquiera dos de las tres entradas como las entradas de selección del multiplexor. Elegimos w_1 y w_2 para tal propósito, lo que resulta en el circuito de la figura 6.8*b*.



a) Implementación con un multiplexor cuatro a uno



b) Tabla de verdad modificada



c) Circuito

Figura 6.7 Síntesis de una función lógica mediante multiplexores.

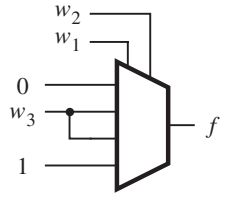
Ejemplo 6.4 En la figura 6.9*a* se indica cómo implementar la función $f = w_1 \oplus w_2 \oplus w_3$ con multiplexores dos a uno. Cuando $w_1 = 0$, f es igual a la XOR de w_2 y w_3, y cuando $w_1 = 1$, f es la XNOR de w_2 y w_3. El multiplexor de la izquierda en el circuito produce $w_2 \oplus w_3$, usando el resultado de la figura 6.7, y el multiplexor derecho emplea el valor de w_1 para elegir $w_2 \oplus w_3$, o su complemento. Note que este circuito pudo haberse derivado directamente escribiendo la función como $f = (w_2 \oplus w_3) \oplus w_1$.

En la figura 6.10 se brinda una implementación de la función XOR de tres entradas con un multiplexor cuatro a uno. La elección de w_1 y w_2 como las entradas de selección resulta en el circuito mostrado.

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

a) Tabla de verdad modificada

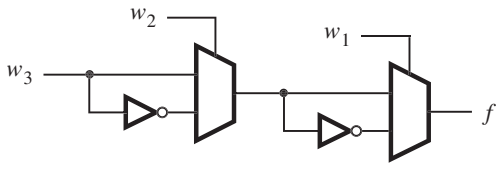


b) Circuito

Figura 6.8 Implementación de la función original de tres entradas con un multiplexor cuatro a uno.

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	$w_2 \oplus w_3$
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	$\overline{w_2 \oplus w_3}$
1	1	0	0	
1	1	1	1	

a) Tabla de verdad

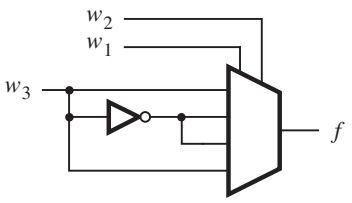


b) Circuito

Figura 6.9 Implementación de XOR de tres entradas con multiplexores dos a uno.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

a) Tabla de verdad



b) Circuito

Figura 6.10 Implementación de XOR de tres entradas con un multiplexor cuatro a uno.

6.1.2 SÍNTESIS DE MULTIPLEXORES MEDIANTE LA EXPANSIÓN DE SHANNON

En las figuras 6.8 a 6.10 se ilustra cómo interpretar las tablas de verdad para implementar funciones lógicas mediante multiplexores. En cada caso, las entradas a los multiplexores son las constantes 0 y 1, o alguna variable o su complemento. Además de usar tales entradas simples, es posible conectar circuitos más complejos como entradas a un multiplexor, lo que permite que las funciones se sinteticen mediante una combinación de multiplexores y otras compuertas lógicas. Supóngase que queremos implementar de esta forma la función original de tres entradas de la figura 6.8 con un multiplexor dos a uno. En la figura 6.11 se muestra una forma intuitiva de realizar esta función. La tabla de verdad puede modificarse como se muestra a la derecha. Si $w_1 = 0$, entonces $f = w_2w_3$, y si $w_1 = 1$, entonces $f = w_2 + w_3$. El uso de w_1 como la entrada de selección para un multiplexor dos a uno conduce al circuito de la figura 6.11*b*.

Esta implementación puede derivarse del modo siguiente usando manipulación algebraica. La función de la figura 6.11*a* se expresa en forma de suma de productos como

$$f = \overline{w}_1w_2w_3 + w_1\overline{w}_2w_3 + w_1w_2\overline{w}_3 + w_1w_2w_3$$

Puede manipularse en

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

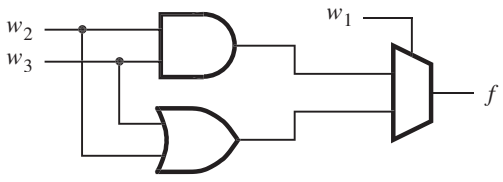
que corresponde al circuito de la figura 6.11*b*.

Las implementaciones de funciones lógicas con multiplexores requieren que una función se descomponga en términos de las variables empleadas como entradas de selección. Esto se logra mediante un teorema propuesto por Claude Shannon [1].

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	w_2w_3
1	$w_2 + w_3$

a) Tabla de verdad



b) Circuito

Figura 6.11 Función original de tres entradas implementada con un multiplexor dos a uno.

Teorema de expansión de Shannon

Cualquier función booleana $f(w_1, \ldots, w_n)$ puede escribirse en la forma

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

Esta expansión puede efectuarse en términos de cualquiera de las n variables. La demostración de este teorema se deja como ejercicio al lector (véase el problema 6.9).

Para ilustrar su uso podemos aplicar el teorema a la función original de tres entradas, que puede escribirse como

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

La expansión de esta función en términos de w_1 produce

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

que es la expresión que derivamos líneas arriba.

Para la función XOR de tres entradas se tiene

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3}) \end{aligned}$$

que produce el circuito de la figura 6.9*b*.

En la expansión de Shannon, el término $f(0, w_2, \ldots, w_n)$ se llama *cofactor* de f respecto a $\overline{w}_1$; se denota con la notación abreviada $f_{\overline{w}_1}$. De manera similar, el término $f(1, w_2, \ldots, w_n)$ se denomina cofactor de f respecto a w_1, y se escribe f_w. Por tanto, es posible escribir

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

En general, si la expansión se realiza respecto de la variable w_i, entonces f_{w_i} denota $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$ y

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

La complejidad de la expresión lógica puede variar, según cuál variable, w_i, se utilice, como se ilustra en el ejemplo 6.5.

Ejemplo 6.5 Para la función $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, la descomposición usando w_1 produce

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

El empleo de w_2 en lugar de w_1 produce

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finalmente, al utilizar w_3 se obtiene

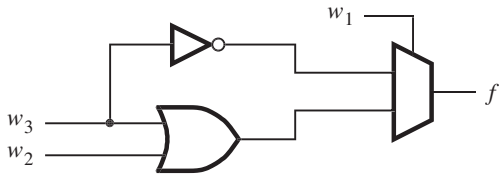
$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

Los resultados generados con w_1 y w_2 tienen el mismo costo, pero el de la expresión producida con el uso de w_3 es más bajo. En la práctica, las herramientas CAD que realizan descomposiciones de este tipo prueban varias posibilidades y eligen la que produce el mejor resultado.

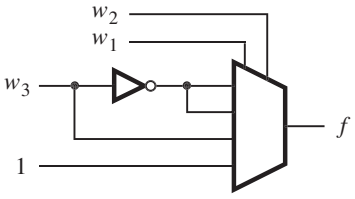
La expansión de Shannon puede efectuarse en términos de más de una variable. Por ejemplo, expandir una función en términos de w_1 y w_2 produce

$$\begin{aligned} f(w_1, \ldots, w_n) = {} & \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) \\ & + w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

Esta expansión genera una forma que puede implementarse con un multiplexor cuatro a uno. Si la expansión de Shannon se realiza en términos de las n variables, entonces el resultado es la forma canónica en suma de productos que definimos en la sección 2.6.1.



a) Con un multiplexor dos a uno



b) Con un multiplexor cuatro a uno

Figura 6.12 Circuitos sintetizados en el ejemplo 6.6.

Ejemplo 6.6

Suponga que se desea implementar la función

$$f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$$

con un multiplexor dos a uno y cualesquiera otras compuertas necesarias. La expansión de Shannon usando w_1 produce

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

El circuito correspondiente se muestra en la figura 6.12*a*. Suponga ahora que se quiere utilizar un multiplexor cuatro a uno. Una mayor descomposición con w_2 produce

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1w_2 f_{w_1w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1w_2(1) \end{aligned}$$

El circuito se muestra en la figura 6.12*b*.

Ejemplo 6.7

Considere la función principal de tres entradas

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

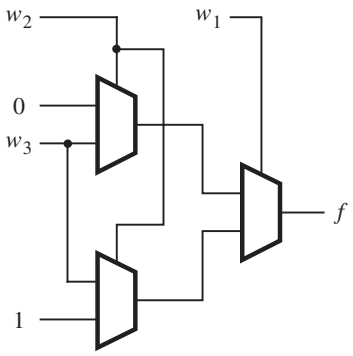


Figura 6.13 Circuito sintetizado en el ejemplo 6.7.

Se quiere implementar esta función usando sólo multiplexores dos a uno. La expansión de Shannon con w_1 produce

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Sean $g = w_2w_3$ y $h = w_2 + w_3$. La expansión de g y h con w_2 produce

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

El circuito correspondiente se muestra en la figura 6.13. Es equivalente al circuito multiplexor cuatro a uno derivado mediante una tabla de verdad en la figura 6.8.

Ejemplo 6.8 En la sección 3.6.5 dijimos que la mayoría de los FPGA emplea tablas de consulta para sus bloques lógicos. Suponga que existe un FPGA en el que cada bloque lógico es una tabla de consulta de tres entradas (LUT 3). Puesto que almacena una tabla de verdad, una LUT 3 puede realizar cualquier función lógica de tres variables. Mediante el uso de la expansión de Shannon puede realizarse cualquier función de cuatro variables con tres LUT 3 a lo sumo. Considere la función

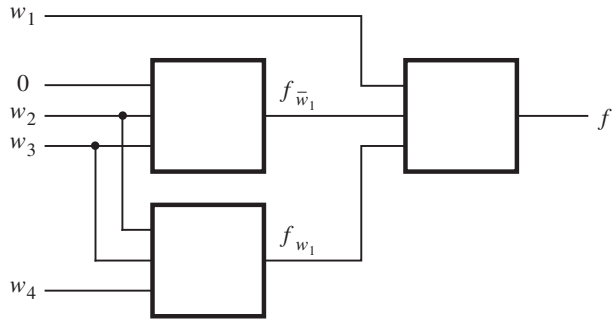
$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

La expansión en términos de w_1 produce

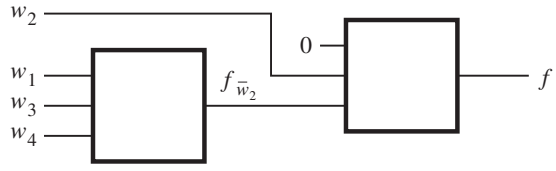
$$\begin{aligned} f &= \overline{w}_1f_{\overline{w}_1} + w_1f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

En la figura 6.14*a* se muestra un circuito con tres LUT 3 que implementa esta expansión. La descomposición de la función utilizando w_2 en lugar de w_1 produce

$$\begin{aligned} f &= \overline{w}_2f_{\overline{w}_2} + w_2f_{w_2} \\ &= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3w_4) \end{aligned}$$



a) Con tres LUT 3



b) Con dos LUT 3

Figura 6.14 Circuitos sintetizados en el ejemplo 6.8.

Observe que $\overline{f}_{\overline{w}_2} = f_{w_2}$; por tanto, sólo se necesitan dos LUT 3, como se ilustra en la figura 6.14*b*. La LUT de la izquierda implementa la función de dos variables $\overline{w}_2 f_{\overline{w}_2} + w_2 \overline{f}_{\overline{w}_2}$.

Como es posible implementar cualquier función lógica con multiplexores, hay chips de uso general que contienen multiplexores como sus recursos lógicos básicos. Tanto Actel Corporation [2] como QuickLogic Corporation [3] ofrecen FPGA en los que el bloque lógico comprende un ordenamiento de multiplexores. Texas Instruments ofrece chips de arreglos de compuertas que tienen bloques lógicos basados en multiplexores [4].

6.2 DECODIFICADORES

Los circuitos decodificadores sirven, valga la redundancia, para decodificar información codificada. Un decodificador binario, descrito en la figura 6.15, es un circuito lógico con n entradas y 2^n salidas. Sólo una salida es válida a la vez, y cada salida corresponde a una combinación de las entradas. El decodificador también tiene una entrada de habilitación (*enable*). *En*, que se utiliza para inhabilitar las salidas; si $En = 0$, entonces ninguna de las salidas es válida. Si $En = 1$, la combinación de $w_{n-1} \cdots w_1 w_0$ determina cuál de las salidas es válida. Un código binario de n bits en el que exactamente uno de los bits se establece en 1 a la vez se llama *codificador de un*

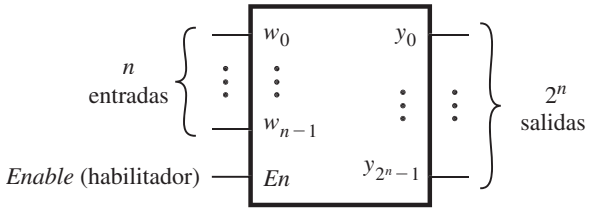


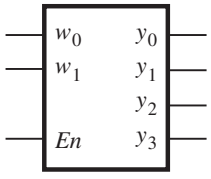
Figura 6.15 Decodificador binario de n a 2^n.

activo, lo que significa que el bit que se establece en 1 se considera "único". Las salidas de un decodificador binario son codificadas en un activo.

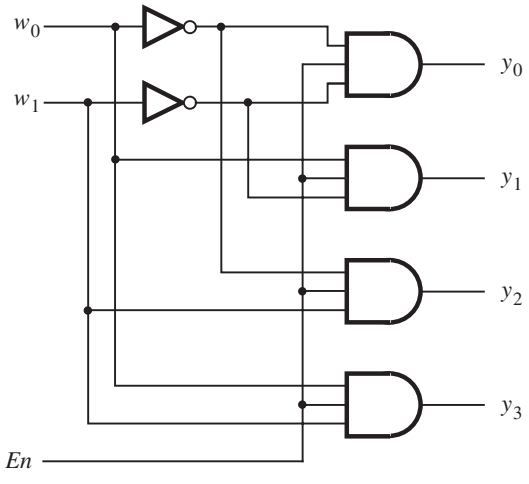
En la figura 6.16 se muestra un decodificador dos a cuatro. Las dos entradas de datos son w_1 y w_0; representan un número de dos bits que hace que el decodificador active una de las salidas $y_0, \ldots, y_3$. Si bien es posible diseñar un decodificador que tenga salidas activa alta o activa baja, en la figura 6.16 se suponen salidas activas altas. Establecer las entradas w_1w_0 en 00, 01,

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

a) Tabla de verdad



b) Símbolo gráfico



c) Circuito lógico

Figura 6.16 Decodificador dos a cuatro.

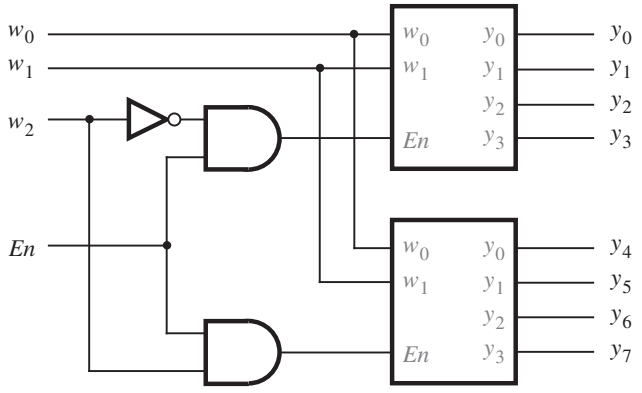


Figura 6.17 Decodificador tres a ocho que usa dos decodificadores dos a cuatro.

10 u 11 ocasiona que la salida y_0, y_1, y_2 o y_3 se establezcan en 1, respectivamente. En el inciso *b*) de la figura se observa un símbolo gráfico para el decodificador, y en el inciso *c*) se muestra un circuito lógico.

Pueden construirse decodificadores más grandes con la estructura de suma de productos de la figura 6.16*c*, o también a partir de decodificadores más pequeños. En la figura 6.17 se ilustra cómo se elabora un decodificador tres a ocho con dos decodificadores dos a cuatro. La entrada w_2 activa las entradas de habilitación de los dos decodificadores. El decodificador superior se habilita si $w_2 = 0$, y el inferior, si $w_2 = 1$. Este concepto se aplica a decodificadores de cualquier tamaño. En la figura 6.18 se indica cómo usar decodificadores dos a cuatro para construir un decodificador 4 a 16. Por su estructura arborescente, este tipo de circuito se llama *árbol decodificador*.

Ejemplo 6.9

Los decodificadores son útiles para muchos fines prácticos. En la figura 6.2*c* mostramos la implementación en suma de productos del multiplexor cuatro a uno, que requiere compuertas AND para distinguir las cuatro combinaciones de las entradas de selección s_1 y s_0. Puesto que un decodificador evalúa los valores en sus entradas, puede emplearse para construir un multiplexor, como se ilustra en la figura 6.19. En este caso no se necesita la entrada de habilitación del decodificador, y se establece en 1. Las cuatro salidas del decodificador representan las cuatro combinaciones de las entradas de selección.

Ejemplo 6.10

En la figura 3.59 mostramos cómo construir un multiplexor dos a uno con dos amortiguadores triestado. Este concepto se aplica a cualquier tamaño de multiplexor, con la adición de un decodificador. En la figura 6.20 se muestra un ejemplo. El decodificador habilita uno de los amortiguadores triestado por cada combinación de las líneas de selección, y ese amortiguador triestado activa la salida, *f*, con la entrada de datos seleccionada. Ahora sabemos que los multiplexores pueden implementarse de varias formas. La elección de emplear la forma en suma de productos, compuertas de transmisión o amortiguadores triestado depende de los recursos disponibles en el chip que se utilizará. Por ejemplo, la mayor parte de los FPGA que utilizan tablas de consulta

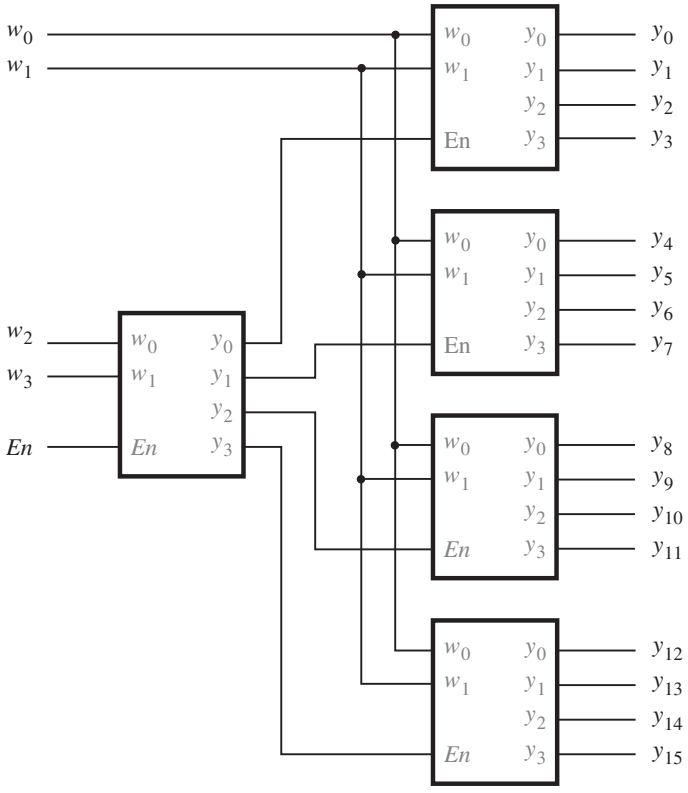


Figura 6.18 Decodificador 4 a 16 construido con un árbol decodificador.

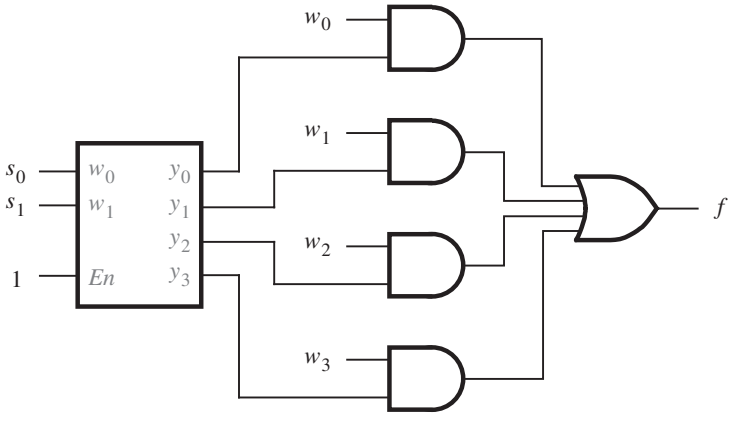


Figura 6.19 Multiplexor cuatro a uno construido con un decodificador.

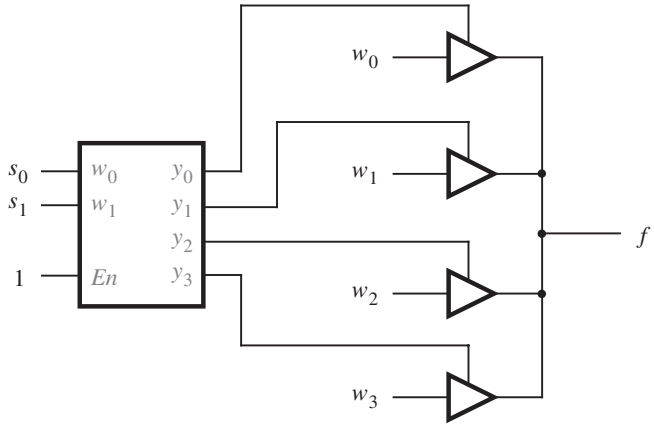


Figura 6.20 Multiplexor cuatro a uno construido con un decodificador y amortiguadores triestado.

para sus bloques lógicos no contiene amortiguadores triestado. En consecuencia, los multiplexores deben implementarse en la forma de suma de productos usando tablas de consulta (véase el ejemplo 6.30).

6.2.1 DEMULTIPLEXORES

En la sección 6.1 mostramos que un multiplexor tiene una salida, n entradas de datos y $\lceil \log_2 n \rceil$ entradas de selección. El propósito del circuito multiplexor es *multiplexar* las n entradas de datos en la salida de datos bajo el control de las entradas de selección. Un circuito que realiza la función opuesta, es decir, que coloca el valor de una sola entrada de datos en varias salidas, se llama *demultiplexor*. El demultiplexor puede implementarse con un circuito decodificador. Por ejemplo, el decodificador dos a cuatro de la figura 6.16 puede usarse como un demultiplexor uno a cuatro. En este caso, la entrada *En* funciona como la entrada de datos para el demultiplexor, y las salidas y_0 a y_3 son las salidas de datos. La combinación de w_1w_0 determina cuál de las salidas se establecerá al valor de *En*. Para ver cómo funciona el circuito, considérese la tabla de verdad de la figura 6.16*a*. Cuando $En = 0$, todas las salidas se establecen en 0, aun la seleccionada por la combinación de w_1w_0. Cuando $En = 1$, la combinación de w_1w_0 establece la salida apropiada en 1.

En general, un circuito decodificador n a 2^n puede usarse como un demultiplexor uno a n. Sin embargo, en la práctica los circuitos decodificadores se utilizan mucho más como decodificadores y no cómo demultiplexores. En muchas aplicaciones, en realidad la entrada *En* del decodificador no es necesaria; por tanto, puede omitirse. En este caso, el decodificador siempre hace válida una de sus salidas de datos, $y_0, \ldots, y_{2^n-1}$, de acuerdo con la combinación de las entradas de datos, $w_{n-1} \cdots w_0$. En el ejemplo 6.11 se usa un decodificador que no tiene la entrada *En*.

Ejemplo 6.11 Una de las aplicaciones más importantes de los decodificadores está en los bloques de memoria, que se utilizan para almacenar información. Tales bloques de memoria se incluyen en los sistemas digitales, como las computadoras, donde es necesario almacenar electrónicamente grandes cantidades de información. Un tipo de bloque de memoria se llama *memoria de sólo lectura* (ROM, *read-only memory*). Una ROM consta de un conjunto de celdas de almacenamiento, y cada una de ellas guarda de manera permanente un solo valor lógico, 0 o 1. En la figura 6.21 se muestra un ejemplo de un bloque ROM. Las celdas de almacenamiento están ordenadas en 2^m filas con n celdas por cada una de ellas. En consecuencia, cada fila almacena n bits de datos. La ubicación de cada fila en la ROM se identifica mediante su *Address* (dirección). En la figura, la fila superior de la ROM tiene dirección 0, y la inferior $2^m - 1$. Puede accederse a la información almacenada en las filas haciendo válidas las líneas de selección, Sel_0 a Sel_{2^m-1}. Como se muestra en la figura, se emplea un decodificador con m entradas y 2^m salidas para generar las señales en las líneas de selección. Como las entradas al decodificador eligen la dirección (fila) en particular seleccionada, se llaman líneas de *dirección*. La información guardada en la fila aparece en las salidas de datos de la ROM, $d_{n-1}, \ldots, d_0$, las cuales se denominan líneas de *datos*. En la figura 6.21 se advierte que cada línea de *datos* tiene un amortiguador triestado asociado que se habilita mediante la entrada de la ROM llamada *Read* (leer). Para tener acceso o, como se dice, *leer* los datos de la ROM, la dirección de la fila deseada se coloca en las líneas de dirección y *Read* se establece en 1.

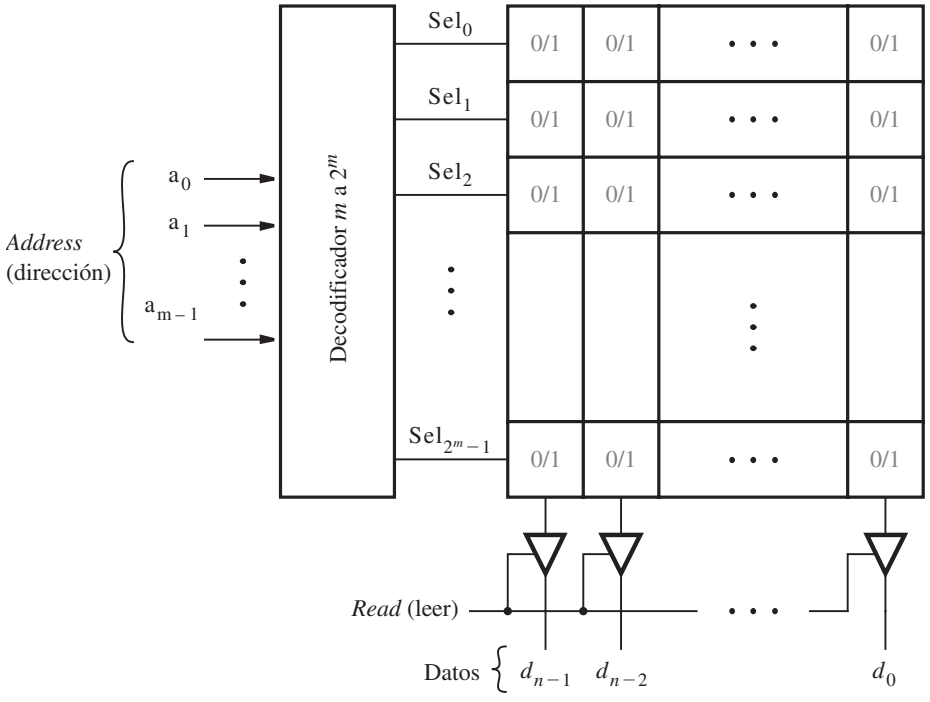


Figura 6.21 Bloque de memoria de sólo lectura (ROM) $2^m \times n$.

Hay muchos tipos de bloques de memoria. En una ROM la información guardada puede leerse de las celdas de almacenamiento, pero no se puede cambiar (véase el problema 6.32). Otro tipo de ROM permite que la información se lea de las celdas de almacenamiento y se almacene, o *escriba* en ellas. Le lectura de su contenido es la operación normal, en tanto que la escritura requiere un procedimiento especial. Tal bloque de memoria recibe el nombre de ROM programable (PROM). Las celdas de almacenamiento en una PROM en general se implementan con transistores EEPROM, que se explicaron en la sección 3.10 para mostrar cómo se usan en los PLD. En la sección 10.1 se analizan otros tipos de bloques de memoria.

6.3 CODIFICADORES

Un codificador realiza la función opuesta de un decodificador. Codifica la información dada en una forma más compacta.

6.3.1 CODIFICADORES BINARIOS

Un *codificador binario* codifica la información de 2^n entradas en un código de n bits, como se indica en la figura 6.22. Exactamente una de las señales de entrada debe tener un valor de 1, y las salidas presentan el número binario que identifica qué entrada es igual a 1. En la figura 6.23*a* se muestra la tabla de verdad para un codificador cuatro a dos. Obsérvese que la salida y_0 es 1 cuando la entrada w_1 o la w_3 es 1, y la salida y_1 es igual a 1 cuando la entrada w_2 o la w_3 es 1. Por tanto, estas salidas pueden generarse mediante el circuito de la figura 6.23*b*. Nótese que hemos supuesto que las entradas son codificadas en 1 activo. En la tabla de verdad no se presentan todos los patrones de entrada que tienen entradas múltiples establecidas en 1, y se tratan como condiciones no importa.

Los codificadores sirven para reducir el número de bits necesarios para representar información específica. Un uso práctico que se les da consiste en transmitir información en un sistema digital. La codificación de los datos permite construir una liga de transmisión empleando menos cables. La codificación también es útil si la información se ha de almacenar para empleo posterior porque se necesitan almacenar menos bits.

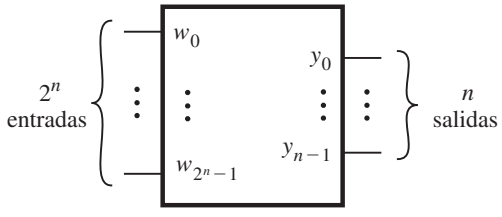
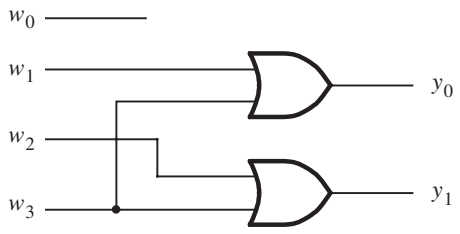


Figura 6.22 Codificador binario 2^n a n.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

a) Tabla de verdad



b) Circuito

Figura 6.23 Codificador binario cuatro a dos.

6.3.2 CODIFICADORES DE PRIORIDAD

Otra clase útil de codificadores se basa en la prioridad de las señales de entrada. En un *codificador de prioridad*, cada entrada tiene un nivel de prioridad asociado con ella. Las salidas del codificador indican la entrada activa que tiene la prioridad más alta. Cuando una entrada se hace válida con una prioridad alta, las demás, con prioridades más bajas, se ignoran. En la figura 6.24 se presenta la tabla de verdad de un codificador de prioridad cuatro a dos. En ella se supone que w_0 tiene la prioridad más baja y w_3 la más alta. Las salidas y_1 y y_0 representan el número binario que identifica la entrada de prioridad más alta establecida en 1. Como es posible que ninguna de las entradas sea igual a 1, se proporciona una salida, z, para indicar esta condición. Se establece en 1 cuando al menos una de las entradas es igual a 1. Se establece en 0 cuando todas las entradas

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figura 6.24 Tabla de verdad para codificador de prioridad cuatro a dos.

son iguales a 0. Las salidas y_1 y y_0 no son significativas en este caso y, por ende, la primera fila de la tabla de verdad puede tratarse como condiciones no importa para y_1 y y_0.

El comportamiento del codificador de prioridad se comprende mejor si primero se considera la última fila de la tabla, donde se especifica que si la entrada w_3 es 1, entonces las salidas se establecen en $y_1 y_0 = 11$. Como w_3 tiene el nivel de prioridad más alto, los valores de las entradas w_2, w_1 y w_0 no importan. Para reflejar que sus valores son irrelevantes, w_2, w_1 y w_0 se denotan mediante el símbolo x en la tabla de verdad. La penúltima fila de la tabla estipula que si $w_2 = 1$, entonces las salidas se establecen en $y_1 y_0 = 10$, pero sólo si $w_3 = 0$. De manera similar, la entrada w_1 hace que las salidas se establezcan en $y_1 y_0 = 01$ sólo si tanto w_3 como w_2 son 0. La entrada w_0 produce las salidas $y_1 y_0 = 00$ sólo si w_0 es la única entrada que se hace válida.

Es posible sintetizar un circuito lógico que implemente la tabla de verdad con las técnicas desarrolladas en el capítulo 4. Sin embargo, una forma más práctica de derivarlo es definir un conjunto de señales intermedias, $i_0, \ldots, i_3$, con base en las observaciones anteriores. Cada señal, i_k, es igual a 1 sólo si la entrada con el mismo índice, w_k, representa la entrada de prioridad más alta que se establece en 1. Las expresiones lógicas para $i_0, \ldots, i_3$ son

$$\begin{aligned} i_0 &= \overline{w}_3 \overline{w}_2 \overline{w}_1 w_0 \\ i_1 &= \overline{w}_3 \overline{w}_2 w_1 \\ i_2 &= \overline{w}_3 w_2 \\ i_3 &= w_3 \end{aligned}$$

Si se utilizan las señales intermedias, el resto del circuito para el codificador de prioridad tiene la misma estructura que el codificador binario de la figura 6.23:

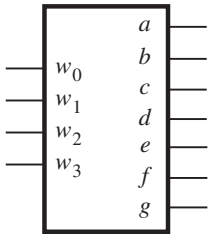
$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \end{aligned}$$

La salida z está dada por

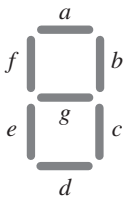
$$z = i_0 + i_1 + i_2 + i_3$$

6.4 CONVERTIDORES DE CÓDIGO

El propósito de los circuitos decodificadores y codificadores es convertir un tipo de codificación de entrada en una codificación de salida diferente. Por ejemplo, un decodificador binario tres a ocho convierte un número binario en la entrada en una codificación de un 1 activo a la salida. Un codificador binario ocho a tres realiza la conversión opuesta. Existen muchos otros tipos posibles de convertidores de código. Un ejemplo común es un decodificador BCD a siete segmentos, que convierte un dígito decimal codificado en binario (BCD) en información adecuada para activar una pantalla orientada a dígitos. Como se ilustra en la figura 6.25*a*, el circuito convierte el dígito BCD en siete señales que se usan para activar los segmentos en la pantalla. Cada segmento es un pequeño diodo emisor de luz (LED, por sus siglas en inglés), que brilla cuando se activa mediante una señal eléctrica. Los segmentos están etiquetados de *a* a *g* en la figura. La tabla de verdad del decodificador BCD a siete segmentos se presenta en la figura 6.25*c*. Para cada combinación de las entradas $w_3, \ldots, w_0$, las sietes salidas se establecen para exhibir el dígito BCD apropiado.



a) Convertidor de código



b) Pantalla de siete segmentos

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

c) Tabla de verdad

Figura 6.25 Convertidor de código BCD a siete segmentos.

Nótese que no se muestran las últimas seis filas de la tabla de verdad completa de 16 filas, ya que representan condiciones no importa porque no son códigos BCD legales y nunca ocurrirán en un circuito que use datos BCD. Es posible derivar un circuito que implemente la tabla de verdad con las técnicas de síntesis expuestas en el capítulo 4. Finalmente, cabe notar que aunque la palabra *decodificador* se usa tradicionalmente para este circuito, un término más apropiado es *convertidor de código*. El término *decodificador* es más adecuado para circuitos que producen salidas que contienen un 1 activo.

6.5 CIRCUITOS DE COMPARACIÓN ARITMÉTICA

En el capítulo 5 expusimos los circuitos aritméticos que realizan sumas, restas y multiplicaciones de números binarios. Otro tipo útil de circuito aritmético compara los tamaños relativos de dos números binarios. Tal circuito se denomina *comparador*. En esta sección consideraremos

el diseño de un comparador que tiene dos entradas de n bits, A y B, que representan números binarios sin signo. El comparador produce tres salidas, llamadas *AeqB*, *AgtB* y *AltB*. La salida *AeqB* se establece en 1 si A y B son iguales. La salida *AgtB* es 1 si A es mayor que B, y la salida *AltB* es 1 cuando A es menor que B.

El comparador deseado puede diseñarse creando una tabla de verdad que especifique las tres salidas como funciones de A y B. Sin embargo, aun para valores moderados de n, la tabla de verdad es grande. Un mejor enfoque consiste en derivar el circuito comparador considerando los bits de A y B en pares. Esto puede ilustrarse mediante un pequeño ejemplo, donde $n = 4$.

Sean $A = a_3a_2a_1a_0$ y $B = b_3b_2b_1b_0$. Defina un conjunto de señales intermedias llamadas i_3, i_2, i_1 e i_0. Cada señal, i_k, es 1 si los bits de A y B con el mismo índice son iguales. Es decir: $i_k = a_k \oplus b_k$. Entonces la salida *AeqB* del comparador está dada por

$$AeqB = i_3i_2i_1i_0$$

Una expresión para la salida *AgtB* puede derivarse si consideramos los bits de A y B en el orden del bit más significativo al menos significativo. La primera posición de bit, k, en la que difieran a_k y b_k determina si A es menor o mayor que B. Si $a_k = 0$ y $b_k = 1$, entonces $A < B$. Pero si $a_k = 1$ y $b_k = 0$, entonces $A > B$. La salida *AgtB* se define por

$$AgtB = a_3\overline{b}_3 + i_3a_2\overline{b}_2 + i_3i_2a_1\overline{b}_1 + i_3i_2i_1a_0\overline{b}_0$$

las señales i_k aseguran que sólo los primeros dígitos de A y B que difieren, considerados de izquierda a derecha, determinan el valor de *AgtB*.

La salida *AltB* puede derivarse con las otras dos salidas como

$$AltB = \overline{AeqB + AgtB}$$

En la figura 6.26 se muestra un circuito lógico que implementa el circuito comparador de cuatro bits. Este enfoque se aplica para diseñar un comparador de cualquier valor de n.

Los circuitos comparadores, como el grueso de los circuitos lógicos, pueden diseñarse de diferentes maneras. Otro enfoque para diseñar un circuito comparador se presenta en el ejemplo 5.10 del capítulo 5.

6.6 VHDL PARA CIRCUITOS COMBINACIONALES

Luego de haber explicado algunos circuitos útiles que pueden emplearse como bloques fundamentales de circuitos más grandes, en las secciones próximas veremos cómo se describen mediante VHDL. Para ello, en vez de basarnos en las instrucciones simples de VHDL usadas en ejemplos anteriores, como en las expresiones lógicas, los especificaremos según su comportamiento. Asimismo, presentaremos varios constructores nuevos de VHDL.

6.6.1 INSTRUCCIONES DE ASIGNACIÓN

VHDL ofrece varios tipos de instrucciones que sirven para asignar valores lógicos a las señales. En los ejemplos de código vistos hasta el momento sólo hemos empleado instrucciones simples de asignación para expresiones lógicas o aritméticas. En esta sección expondremos otros tipos

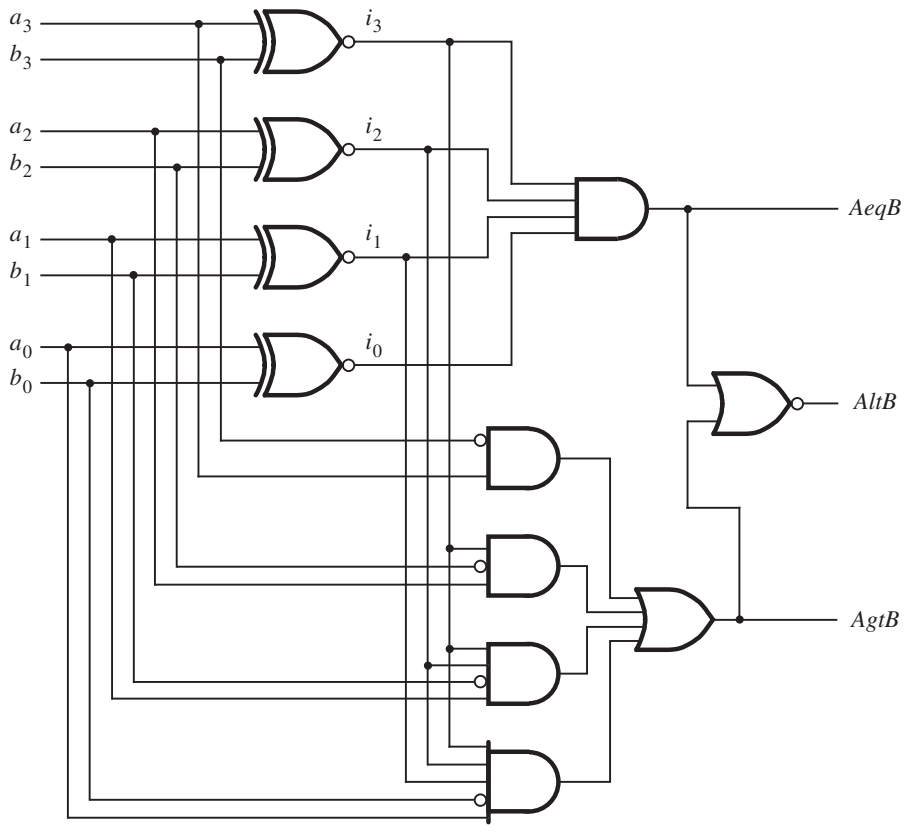


Figura 6.26 Circuito comparador de cuatro bits.

de instrucciones de asignación, llamadas asignaciones de señal seleccionada, asignaciones de señal condicional, instrucciones de generación, instrucciones if-then-else e instrucciones case.

6.6.2 ASIGNACIÓN DE SEÑAL SELECCIONADA

Una asignación de señal seleccionada permite asignar uno de varios valores a una señal, con base en un criterio de selección. En la figura 6.27 se muestra cómo usarla para describir un multiplexor dos a uno. La entidad, llamada *mux2to1*, tiene las entradas w_0, w_1 y s, y la salida f. La asignación de señal seleccionada comienza con la palabra reservada WITH, que especifica que s se usará para el criterio de selección. Las dos cláusulas WHEN establecen que se asigna a f el valor de w_0 cuando $s = 0$; de otro modo, se le asigna el valor de w_1. La cláusula WHEN que selecciona w_1 utiliza la palabra OTHERS, en lugar del valor 1. Esto es necesario porque la sintaxis de VHDL indica que hay que incluir una cláusula WHEN para todo posible valor de la señal de

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <=  w0 WHEN '0',
              w1 WHEN OTHERS ;
END Behavior ;
```

Figura 6.27 Código de VHDL para un multiplexor dos a uno.

selección *s*. Como contiene el tipo STD_LOGIC, explicado en la sección 4.12, *s* puede tomar los valores 0, 1, Z, − y otros. La palabra reservada OTHERS ofrece una forma práctica de explicar todos los valores lógicos que no se enumeran explícitamente en una cláusula WHEN.

Ejemplo 6.12

Un multiplexor cuatro a uno se describe mediante la entidad llamada *mux4to1*, que se muestra en la figura 6.28. Las dos entradas de selección, que en la figura 6.2 se denominan s_1 y s_0, están representadas por la señal *s* de dos bits STD_LOGIC_VECTOR. La asignación de señal seleccionada establece *f* al valor de una de las entradas w_0 , . . . , w_3, según la combinación de *s*. La compilación del código resulta en el circuito que aparece en la figura 6.2*c*. Al final de la figura 6.28, la entidad *mux4to1* se define como un componente del paquete llamado *mux4to1_package*. En la sección 5.5.2 mostramos que la declaración de componente permite utilizar la entidad como un subcircuito en otro código de VHDL.

Ejemplo 6.13

En la figura 6.4 se mostró cómo construir un multiplexor 16 a 1 con cinco multiplexores cuatro a uno. En la figura 6.29 se presenta el código de VHDL para ese circuito, empleando el componente *mux4to1*. Las líneas de código se numeran de modo que nos podamos remitir fácilmente a ellas. El paquete *mux4to1_package* se incluye en el código, puesto que ofrece la declaración de componente para *mux4to1*.

Las entradas de datos a la entidad *mux16to1* es la señal de 16 bits llamada *w*, y las entradas de selección es la señal de cuatro bits llamada *s*. En el código de VHDL se necesitan nombres de señal para las salidas de los cuatro multiplexores cuatro a uno en el lado izquierdo de la figura 6.4. La línea 11 define una señal de cuatro bits llamada *m* para este propósito, y las líneas 13 a 16 instancian los cuatro multiplexores. Por ejemplo, la línea 13 corresponde al multiplexor de la esquina superior izquierda de la figura 6.4. Sus primeros cuatro puertos, que corresponden a w_0 , . . . , w_3 en la figura 6.28, se activan con las señales *w*(0) , . . . , *w*(3). La sintaxis

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( w0, w1, w2, w3  : IN   STD_LOGIC ;
           s               : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           f               : OUT  STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN "00",
             w1 WHEN "01",
             w2 WHEN "10",
             w3 WHEN OTHERS ;
END Behavior ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT ( w0, w1, w2, w3  : IN   STD_LOGIC ;
               s               : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               f               : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```

Figura 6.28 Código de VHDL para un multiplexor cuatro a uno.

s(1 DOWNTO 0) se emplea para unir las señales *s*(1) y *s*(0) al puerto *s* de dos bits del componente *mux4to1*. La señal *m*(0) se conecta al puerto de salida del multiplexor.

La línea 17 instancia el multiplexor a la derecha de la figura 6.4. Las señales $m_0, \ldots, m_3$ están conectadas a sus entradas de datos, y los bits *s*(3) y *s*(2), que se especifican mediante la sintaxis *s*(3 DOWNTO 2), se unen a las entradas de selección. El puerto de salida genera la salida *f* de *mux16to1*. Al compilar el código se obtiene la función multiplexora siguiente

$$f = \overline{s}_3\overline{s}_2\overline{s}_1\overline{s}_0 w_0 + \overline{s}_3\overline{s}_2\overline{s}_1 s_0 w_1 + \overline{s}_3\overline{s}_2 s_1\overline{s}_0 w_2 + \cdots + s_3 s_2 s_1\overline{s}_0 w_{14} + s_3 s_2 s_1 s_0 w_{15}$$

Ejemplo 6.14 Las asignaciones de señal seleccionada también pueden usarse para escribir otros tipos de circuitos. En la figura 6.30 se muestra cómo emplear una asignación de señal seleccionada para describir la tabla de verdad de un decodificador binario dos a cuatro. La entidad se designa *dec2to4*.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   LIBRARY work ;
4   USE work.mux4to1_package.all ;

5   ENTITY mux16to1 IS
6       PORT ( w  : IN    STD_LOGIC_VECTOR(0 TO 15) ;
7              s  : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
8              f  : OUT   STD_LOGIC ) ;
9   END mux16to1 ;

10  ARCHITECTURE Structure OF mux16to1 IS
11      SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
12  BEGIN
13      Mux1: mux4to1 PORT MAP
            ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
14      Mux2: mux4to1 PORT MAP
            ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
15      Mux3: mux4to1 PORT MAP
            ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
16      Mux4: mux4to1 PORT MAP
            ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
17      Mux5: mux4to1 PORT MAP
            ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
18  END Structure ;
```

Figura 6.29 Código jerárquico para un multiplexor 16 a 1.

Las entradas de datos son la señal de dos bits llamada *w*, y la entrada de habilitación es *En*. Las cuatro salidas se representan mediante la señal de cuatro bits *y*.

En la tabla de verdad para el decodificador de la figura 6.16*a*, las entradas se enumeran en el orden *En* w_1w_0. Para representar estas tres señales, el código de VHDL define la señal de tres bits llamada *Enw*. La instrucción *Enw* <= *En* & *w* usa el operador de concatenación de VHDL (&), expuesto en la sección 5.5.4, para combinar las señales *En* y *w* en la señal *Enw*. Por tanto, $Enw(2) = En$, $Enw(1) = w_1$, y $Enw(0) = w_0$. La señal *Enw* se utiliza como la señal de selección en la instrucción de asignación de señal seleccionada. Describe la tabla de verdad de la figura 6.16*a*. En las primeras cuatro cláusulas WHEN, $En = 1$, y las salidas del decodificador tienen los mismos patrones que en las primeras cuatro filas de la tabla de verdad. La última cláusula WHEN emplea la palabra reservada OTHERS y establece las salidas del decodificador en 0000 porque representa los casos en que $En = 0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;
END Behavior ;
```

Figura 6.30 Código de VHDL para un decodificador binario dos a cuatro.

6.6.3 ASIGNACIÓN DE SEÑAL CONDICIONAL

Similar a la asignación de señal seleccionada, la asignación de señal condicional permite que una señal se establezca en uno de varios valores. En la figura 6.31 se muestra una versión modificada de la entidad multiplexor dos a uno de la figura 6.27. Utiliza una asignación de señal condicional para especificar que se asigna a f el valor de w_0 cuando $s = 0$; de otro modo, se le asigna el valor de w_1. Al compilar el código se genera el mismo circuito que el del código de la

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

Figura 6.31 Especificación de un multiplexor dos a uno con una asignación de señal condicional.

figura 6.27. En este pequeño ejemplo, la asignación de señal condicional tiene sólo una cláusula WHEN. Un ejemplo más complejo, que ilustra mejor las características de la asignación de señal condicional, se proporciona en el ejemplo 6.15.

Ejemplo 6.15

En la figura 6.24 se muestra la tabla de verdad de un codificador de prioridad cuatro a dos. En la figura 6.32 se presenta el código de VHDL que describe esta tabla. Las entradas al codificador se representan mediante la señal de cuatro bits llamada *w*. El codificador tiene las salidas *y*, que es una señal de dos bits, y *z*.

La asignación de señal condicional especifica que a *y* se le asigna el valor 11 cuando la entrada $w(3) = 1$. Si esta condición es verdadera, entonces las demás cláusulas WHEN que vienen después de la palabra reservada ELSE no afectan el valor de *f*. Por tanto, los valores de $w(2)$, $w(1)$ y $w(0)$ no importan, lo que implementa el esquema de prioridad deseado. La segunda cláusula WHEN establece que cuando $w(2) = 1$, entonces se asigna el valor 10 a *y*. Esto sólo puede ocurrir si $w(3) = 0$. Cada cláusula WHEN sucesiva puede afectar a *y* sólo si ninguna de las condiciones asociadas con las cláusulas WHEN precedentes es verdadera. En la figura 6.32 se incluye una segunda asignación de señal condicional para la salida *z*, que afirma que cuando las cuatro entradas son 0, se asigna a *z* el valor 0; de otro modo, se le asigna el valor 1.

El nivel de prioridad asociado con cada cláusula WHEN en la asignación de señal condicional es una diferencia principal con la asignación de señal seleccionada, que no tiene tal esquema de prioridad. El codificador de prioridad puede describirse con una asignación de señal seleccionada, pero el código es más complicado. Una posibilidad se muestra con la arquitectura presentada en la figura 6.33. La primera cláusula WHEN establece *y* en 00 cuando w_0 es la única entrada igual a 1. Las dos cláusulas siguientes determinan que *y* debe ser 01 cuando $w_3 = w_2 = 0$ y $w_1 = 1$. Las cuatro cláusulas siguientes especifican que *y* debe ser 10 si $w_3 = 0$ y $w_2 = 1$. Finalmente, la última cláusula WHEN establece que *y* debe ser 1 para todas las demás combinaciones de entrada, lo que incluye las combinaciones para las que w_3 es 1. Nótese que las

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT    STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <= "11" WHEN w(3) = '1' ELSE
         "10" WHEN w(2) = '1' ELSE
         "01" WHEN w(1) = '1' ELSE
         "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figura 6.32 Código de VHDL para un codificador de prioridad.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    WITH w SELECT
        y <= ”00” WHEN ”0001”,
             ”01” WHEN ”0010”,
             ”01” WHEN ”0011”,
             ”10” WHEN ”0100”,
             ”10” WHEN ”0101”,
             ”10” WHEN ”0110”,
             ”10” WHEN ”0111”,
             ”11” WHEN OTHERS ;
    WITH w SELECT
        z <= ’0’ WHEN ”0000”,
             ’1’ WHEN OTHERS ;
END Behavior ;
```

Figura 6.33 Código menos eficiente para un codificador de prioridad.

cláusulas OTHERS comprenden la combinación de entrada 0000. Este patrón resulta en $z = 0$, y el valor de y no importa en este caso.

Ejemplo 6.16 En la figura 6.26 derivamos el circuito para un comparador. En la figura 6.34 se muestra cómo describirlo con código de VHDL. Cada una de las tres asignaciones de señal condicional determinan el valor de una de las salidas del comparador. El paquete llamado *std_logic_unsigned* se incluye en el código porque indica que las señales STD_LOGIC_VECTOR, designadas *A* y *B*, pueden usarse como números binarios sin signo con operadores relacionales de VHDL. Estos últimos ofrecen una forma práctica de especificar la funcionalidad deseada.

El circuito generado a partir del código de la figura 6.34 es similar, pero no idéntico, al de la figura 6.26. El compilador de VHDL instancia un módulo predefinido para implementar cada operación de comparación. En Quartus II, los módulos que se instancian son de la biblioteca LPM, que expusimos en la sección 5.5.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY compare IS
    PORT ( A, B                 : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           AeqB, AgtB, AltB     : OUT   STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figura 6.34 Código de VHDL para un comparador de cuatro bits.

En vez de utilizar la biblioteca *std_logic_unsigned*, otra forma de especificar que el circuito generado debe usar números sin signo consiste en incluir la biblioteca llamada *std_logic_arith*. En este caso, las señales *A* y *B* deben definirse con el tipo UNSIGNED, en lugar de STD_LOGIC_VECTOR. Si se quiere que el circuito trabaje con números con signo, las señales *A* y *B* han de definirse con el tipo SIGNED. Este código se presenta en la figura 6.35.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY compare IS
    PORT ( A, B                 : IN    SIGNED(3 DOWNTO 0) ;
           AeqB, AgtB, AltB     : OUT   STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Figura 6.35 Código de la figura 6.34 para números con signo.

6.6.4 INSTRUCCIONES DE GENERACIÓN

En la figura 6.29 se muestra el código de VHDL para un multiplexor 16 a 1 que utiliza cinco instancias de un subcircuito multiplexor cuatro a uno. La estructura regular del código sugiere que se podría escribir en una forma más compacta empleando un ciclo. VHDL incluye una instrucción llamada FOR GENERATE para describir código jerárquico estructurado regularmente.

En la figura 6.36 se muestra el código de la figura 6.29 reescrito empleando una instrucción FOR GENERATE. Como ésta debe tener una etiqueta, en el código se utiliza *G1*. El ciclo instancia cuatro copias del componente *mux4to1*, mediante el índice *i* del ciclo en el límite de 0 a 3. La variable *i* no se declara explícitamente en el código; se define de forma automática como una variable local cuyo alcance se limita a la instrucción FOR GENERATE. La primera iteración del ciclo corresponde a la instrucción de instanciación etiquetada *Mux1* en la figura 6.29. El operador * representa multiplicación; por tanto, en la primera iteración del ciclo el compilador de VHDL traduce los nombres de señal $w(4 * i)$, $w(4 * i + 1)$, $w(4 * i + 2)$ y $w(4 * i + 3)$ en nombres de señal $w(0)$, $w(1)$, $w(2)$ y $w(3)$. Las iteraciones del ciclo para $i = 1$, $i = 2$ e $i = 3$ corresponden a las instrucciones etiquetadas *Mux2*, *Mux3* y *Mux4* en la figura 6.29. Como la instrucción etiquetada *Mux5* en la figura 6.29 no va en el ciclo, se incluye por separado en la figura 6.36. El circuito generado a partir del código de la figura 6.36 es idéntico al producido por el código de la figura 6.29.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT ( w  : IN   STD_LOGIC_VECTOR(0 TO 15) ;
           s  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f  : OUT  STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Muxes: mux4to1 PORT MAP (
            w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
    END GENERATE ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Figura 6.36 Código para un multiplexor 16 a 1 que usa una instrucción de generación.

Ejemplo 6.17

Además de la instrucción FOR GENERATE, VHDL ofrece otro tipo de instrucción similar llamada IF GENERATE. En la figura 6.37 se ilustra el uso de las dos. El código presentado es una descripción jerárquica del decodificador 4 a 16 mostrado en la figura 6.18, usando cinco instancias del componente *dec2to4* definido en la figura 6.30. Las entradas del decodificador son la señal de cuatro bits *w*, el habilitador es *En* y las salidas son la señal de 16 bits *y*.

Después de la declaración de componente para el subcircuito *dec2to4*, la arquitectura define la señal *m*, que representa las salidas del decodificador dos a cuatro del lado izquierdo de la figura 6.18. Las cinco copias del componente *dec2to4* se instancian mediante la instrucción FOR GENERATE. En cada iteración del ciclo, la instrucción etiquetada *Dec_ri* instancia un componente *dec2to4* que corresponde a uno de los decodificadores dos a cuatro en el lado derecho de la figura 6.18. La primera iteración del ciclo genera el componente *dec2to4* con entradas de datos w_1 y w_0, entrada de habilitación m_0 y salidas y_0, y_1, y_2, y_3. Las otras iteraciones del ciclo también utilizan entradas de datos w_1w_0, pero diferentes bits de *m* y *y*.

La instrucción IF GENERATE, etiquetada *G2*, instancia un componente *dec2to4* en la última iteración de ciclo, para el que la condición $i = 3$ es verdadera. Este componente representa el decodificador dos a cuatro del lado izquierdo de la figura 6.18. Tiene las entradas de datos de dos bits w_3 y w_2, el habilitador *En* y las salidas m_0, m_1, m_2 y m_3. Nótese que en vez de usar la

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 15) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
    COMPONENT dec2to4
        PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En  : IN    STD_LOGIC ;
               y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
    END COMPONENT ;
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    G1: FOR i IN 0 TO 3 GENERATE
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i+3) );
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(i DOWNTO i-1), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```

Figura 6.37 Código jerárquico para un decodificador binario 4 a 16.

instrucción IF GENERATE pudimos haber instanciado este componente fuera de la instrucción FOR GENERATE. Escribimos el código tal cual se muestra simplemente para dar un ejemplo de la instrucción IF GENERATE.

Las instrucciones de generación de las figuras 6.36 y 6.37 sirven para instanciar componentes. Otro uso consiste en generar un conjunto de ecuaciones lógicas. En la figura 7.73 daremos un ejemplo de ello.

6.6.5 INSTRUCCIONES DE ASIGNACIÓN CONCURRENTE Y SECUENCIAL

Hemos expuesto varios tipos de instrucciones de asignación: de asignación simple, que suponen expresiones lógicas o aritméticas, de asignación seleccionada y de asignación condicional. Todas ellas comparten la propiedad de que su orden de aparición no afecta el significado del código de VHDL. Por ello tales instrucciones reciben el nombre de *instrucciones de asignación concurrentes*.

VHDL también ofrece una segunda categoría de instrucciones, llamadas *instrucciones de asignación secuencial*, cuyo orden sí puede afectar el significado del código. Explicaremos dos tipos de instrucciones de asignación secuencial: las instrucciones if-then-else y las instrucciones case. En VHDL es obligatorio que las instrucciones de asignación secuencial se coloquen dentro de otro tipo de instrucción, la instrucción process.

6.6.6 INSTRUCCIÓN PROCESS

En las figuras 6.27 y 6.31 se muestran dos formas de describir un multiplexor dos a uno con asignaciones de señal seleccionada y condicional. El mismo circuito también puede describirse con una instrucción if-then-else, pero debe colocarse dentro de una instrucción process. En la figura 6.38 se presenta tal código. La instrucción process comienza con la palabra reservada PROCESS, seguida de una lista de señales encerradas entre paréntesis, llamada *lista de sensibilidad*. Para un circuito combinacional como el multiplexor, la lista de sensibilidad incluye todas las señales de entrada que se usan dentro del proceso. El compilador de VHDL traduce la instrucción process en ecuaciones lógicas. En la figura, el proceso consta de una sola instrucción if-then-else que describe la función multiplexor. Por ende, la lista de sensibilidad comprende las entradas de datos, w_0 y w_1, y la entrada de selección, s.

En general, puede haber varias instrucciones dentro de un proceso, las cuales se evalúan como sigue. En la jerga de VHDL se dice que cuando hay un cambio en el valor de cualquier señal en la lista de sensibilidad de proceso, entonces el proceso se vuelve *activo*. Una vez activo, las instrucciones dentro de él se evalúan en orden secuencial. Cualquier asignación hecha a las señales dentro del proceso no es visible fuera de él hasta que todas las instrucciones internas se han evaluado. Si existen varias asignaciones a la misma señal, sólo la última tiene algún efecto visible. Esto se ilustra en el ejemplo 6.18.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.38 Multiplexor dos a uno especificado con la instrucción if-then-else.

Ejemplo 6.18

El código de la figura 6.39 equivale al de la 6.38. La primera instrucción en el proceso asigna el valor de w_0 a f. Esto ofrece un valor *predeterminado* de f, pero la asignación en realidad no tiene lugar hasta el final del proceso. En la jerga de VHDL se dice que la asignación se *programa* para que ocurra después de que se han evaluado todas las instrucciones en el proceso. Si hay otra asignación a f mientras el proceso está activo, la asignación predeterminada se invalida. La segunda instrucción en el proceso asigna a f el valor de w_1 si el valor de s es igual a 1. Si esta condición es verdadera, entonces la asignación predeterminada se invalida. Por tanto, si $s = 0$, entonces $f = w_0$, y si $s = 1$, entonces $f = w_1$, lo que define el multiplexor dos a uno. La compilación de este código resulta en el mismo circuito que el de las figuras 6.27, 6.31 y 6.38: $f = \bar{s}w_0 + sw_1$.

La instrucción process de la figura 6.39 ilustra que el orden de las instrucciones dentro de él puede afectar el significado del código. Considérese invertir el orden de las dos instrucciones de modo que if-then-else se evalúe primero. Si $s = 1$, se asigna a f el valor de w_1. Esta asignación se programa y no tiene lugar hasta el final del proceso. Sin embargo, la instrucción $f <= w_0$ se evalúa al último. Esto invalida la primera asignación, y se asigna a f el valor de w_0 independientemente del valor de s. Por ende, en vez de describir un multiplexor, cuando las instrucciones dentro del proceso se invierten, el código representa el circuito trivial $f = w_0$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN    STD_LOGIC ;
           f          : OUT   STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.39 Código alternativo para el multiplexor dos a uno con una instrucción if-then-else.

Ejemplo 6.19 La figura 6.40 proporciona un ejemplo que contiene tanto una instrucción de asignación concurrente como una de proceso. En la figura se describe un codificador de prioridad equivalente al código de la figura 6.32. El proceso describe el esquema de prioridad deseado con una instrucción if-then-else. Especifica que si la entrada w_3 es 1, entonces la salida se establece en $y = 11$. Esta asignación no depende de los valores de las entradas w_3, w_1 o w_0; en consecuencia, sus valores no importan. Las otras cláusulas en la instrucción if-then-else se evalúan sólo si $w_3 = 0$. La primera cláusula ELSIF indica que si w_2 es 1, entonces $y = 10$. Si $w_2 = 0$, entonces la siguiente cláusula ELSIF resulta en $y = 01$ si $w_1 = 1$. Si $w_3 = w_2 = w_1 = 0$, entonces la cláusula ELSIF resulta en $y = 00$. Esta asignación se realiza ya sea que w_0 sea o no 1; en la figura 6.24 se indica que y puede establecerse en cualquier patrón cuando $w = 0000$ porque z se establecerá en 0 en este caso.

La salida z del codificador de prioridad debe establecerse en 1 siempre que al menos una de las entradas de datos sea 1. Esta salida se define mediante la instrucción de asignación condicional al final de la figura 6.40. La sintaxis de VHDL no permite que dentro de un proceso aparezca una instrucción de asignación condicional (o una de asignación seleccionada). Una posibilidad consiste en especificar el valor de z con una instrucción if-then-else dentro del proceso. La razón por la que escribimos el código como se presenta en la figura es para ilustrar que las instrucciones de asignación concurrente pueden usarse junto con instrucciones process. La instrucción process sirve para separar las instrucciones secuenciales de las concurrentes. Nótese que no importa el orden de la instrucción process y de la de asignación condicional. VHDL estipula que mientras las instrucciones dentro de un proceso sean secuenciales, la instrucción process en sí misma es concurrente.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT    STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) = '1' THEN
            y <= "10" ;
        ELSIF w(1) = '1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <   '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Figura 6.40 Codificador de prioridad especificado con la instrucción if-then-else.

Ejemplo 6.20

En la figura 6.41 se muestra otro estilo de código para el codificador de prioridad; ahora se utilizan instrucciones if-then-else. La primera instrucción en el proceso proporciona el valor predeterminado de 00 para $y_1 y_0$. La segunda lo invalida si w_1 es 1, y establece $y_1 y_0$ en 01. De manera similar, la tercera y cuarta instrucciones invalidan las anteriores si w_2 o w_3 son 1, y establece $y_1 y_0$ en 10 y 11, respectivamente. Estas cuatro instrucciones equivalen a la única instrucción if-then-else de la figura 6.40 que describe el esquema de prioridad. El valor de z se especifica mediante una instrucción de asignación predeterminada, seguida por una instrucción if-then-else que invalida el valor predeterminado si $w = 0000$. Aunque los ejemplos de las figuras 6.40 y 6.41 son equivalentes, es probable que el significado del código de la 6.40 sea más sencillo de entender.

Ejemplo 6.21

En la figura 6.34 se especifica un comparador de cuatro bits que produce las tres salidas *AeqB*, *AgtB* y *AltB*. En la figura 6.42 se muestra cómo tal especificación puede escribirse con instrucciones if-then-else. Por simplicidad, para las entradas *A* y *B* se usan números de un bit, y sólo se

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z  : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        y <= "00" ;
        IF w(1) = '1' THEN y <= "01" ; END IF ;
        IF w(2) = '1' THEN y <= "10" ; END IF ;
        IF w(3) = '1' THEN y <= "11" ; END IF ;

        z <= '1' ;
        IF w = "0000" THEN z <= '0' ; END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.41 Código alternativo para el codificador de prioridad.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B  : IN     STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.42 Código para un comparador de igualdad de un bit.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B  : IN   STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.43 Ejemplo de código que resulta en memoria implícita.

muestra el código para la salida *AeqB*. El proceso asigna el valor predeterminado de 0 a *AeqB* y luego la instrucción if-then-else cambia *AeqB* a 1 si *A* y *B* son iguales. Es útil considerar el efecto en la semántica del código si se elimina la instrucción de asignación predeterminada, como se ilustra en la figura 6.43.

Con sólo la instrucción if-then-else, el código no especifica qué valor debe tener *AeqB* si la condición $A = B$ no es verdadera. La semántica de VHDL estipula que en casos en que el código no especifica el valor de una señal, ésta debe conservar su valor actual. Para el código de la figura 6.43, una vez que *A* y *B* son iguales, lo que resulta en $AeqB = 1$, entonces *AeqB* permanecerá en 1 indefinidamente, aun si *A* y *B* ya no son iguales. En la jerga de VHDL, se dice que la salida *AeqB* tiene *memoria implícita* porque el circuito que se sintetice a partir del código "recordará" o almacenará el valor $AeqB = 1$. En la figura 6.44 se muestra el circuito sintetizado a partir del código. La compuerta XOR produce un 1 cuando *A* y *B* son iguales, y la compuerta OR garantiza que *AeqB* permanezca en 1 de manera indefinida.

La memoria implícita que resulta del código de la figura 6.43 no es útil porque genera un circuito comparador que no funciona correctamente. Sin embargo, en el capítulo 7 demostra-

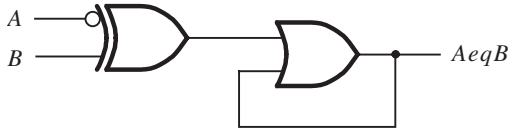


Figura 6.44 Circuito generado a partir del código de la figura 6.43.

remos que la semántica de la memoria implícita es útil para otros tipos de circuitos, los cuales tienen la capacidad de almacenar valores de señal lógica en elementos de memoria.

6.6.7 INSTRUCCIÓN CASE

Una instrucción case es similar a una asignación de señal seleccionada porque tiene una señal de selección e incluye cláusulas WHEN para diversas combinaciones de esa señal. En la figura 6.45 se muestra cómo usar la instrucción case como otra forma de describir el circuito multiplexor dos a uno. La instrucción case comienza con la palabra reservada CASE, que especifica que s se usará como la señal de selección. La primera cláusula WHEN especifica, después del símbolo =>, las instrucciones que han de evaluarse cuando $s = 0$. En este ejemplo, la única instrucción que se evalúa cuando $s = 0$ es $f <= w_0$. La instrucción case debe incluir cláusulas WHEN para todas las posibles combinaciones de la señal de selección. Por tanto, la segunda cláusula WHEN, que contiene $f <= w_1$, usa la palabra reservada OTHERS.

Ejemplo 6.22 En la figura 6.30 se proporciona el código para un decodificador dos a cuatro. Una forma distinta de describir este circuito, mediante instrucciones de asignación secuencial, se muestra en la figura 6.46. Primero, el proceso usa una instrucción if-then-else para comprobar el valor de

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT ( w0, w1, s  : IN   STD_LOGIC ;
           f          : OUT  STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 6.45 Instrucción case que representa un multiplexor dos a uno.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN   STD_LOGIC ;
           y   : OUT  STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>
                    y <= "1000" ;
                WHEN "01" =>
                    y <= "0100" ;
                WHEN "10" =>
                    y <= "0010" ;
                WHEN OTHERS =>
                    y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.46 Instrucción process que describe un decodificador binario dos a cuatro.

la señal habilitadora del decodificador *En*. Si $En = 1$, la instrucción case establece la salida y al valor apropiado con base en la entrada *w*. La instrucción case representa las primeras cuatro filas de la tabla de verdad de la figura 6.16*a*. Si $En = 0$, la cláusula ELSE establece *y* a 0000, como se especifica en la fila inferior de la tabla de verdad.

Ejemplo 6.23

Otro ejemplo de una instrucción case se presenta en la figura 6.47. La entidad se llama *seg7* y representa al decodificador BCD a siete segmentos de la figura 6.25. La entrada BCD se representa mediante la señal de cuatro bits llamada *bcd*, y las siete salidas son la señal de siete bits llamada *leds*. La instrucción case se formatea para que recuerde la tabla de verdad de la figura 6.25*c*. Note que hay un comentario a la derecha de esa instrucción, el cual etiqueta las siete

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
    PORT ( bcd   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds  : OUT   STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                    --    abcdefg
            WHEN "0000"   => leds <= "1111110" ;
            WHEN "0001"   => leds <= "0110000" ;
            WHEN "0010"   => leds <= "1101101" ;
            WHEN "0011"   => leds <= "1111001" ;
            WHEN "0100"   => leds <= "0110011" ;
            WHEN "0101"   => leds <= "1011011" ;
            WHEN "0110"   => leds <= "1011111" ;
            WHEN "0111"   => leds <= "1110000" ;
            WHEN "1000"   => leds <= "1111111" ;
            WHEN "1001"   => leds <= "1110011" ;
            WHEN OTHERS => leds <= "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 6.47 Código que representa el decodificador BCD a siete segmentos.

salidas con las letras de la *a* a la *g*. Dichas etiquetas indican al lector la correlación entre la señal *leds* de siete bits en el código de VHDL y los siete segmentos de la figura 6.25*b*. La cláusula WHEN final de la instrucción case establece los siete bits de *leds* a –. Recuérdese que – se usa en VHDL para denotar una condición no importa. Esta cláusula representa las condiciones no importa explicadas para la figura 6.25, que son los casos donde la entrada *bcd* no representa un dígito BCD válido.

Ejemplo 6.24 Una unidad lógica aritmética (ALU, *arithmetic logic unit*) es un circuito lógico que realiza varias operaciones booleanas y aritméticas con operandos de *n* bits. En la sección 3.5 presentamos una familia de chips estándar llamada chips de la serie 7400. Dijimos que algunos de esos chips contienen compuertas lógicas básicas y otros ofrecen circuitos lógicos usados comúnmente. Un ejemplo de ALU es el chip estándar denominado 74381. En la tabla 6.1 se especifica la funcionalidad de este chip. Tiene dos entradas de datos de cuatro bits, llamadas *A* y *B*; una entrada de

Tabla 6.1 Funcionalidad de la ALU *74381*.

Operación	Entradas $s_2\ s_1\ s_0$	Salidas F
Clear	0 0 0	0 0 0 0
B −A	0 0 1	B −A
A −B	0 1 0	A −B
ADD	0 1 1	A +B
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

selección de tres bits, *s*; y una salida de cuatro bits, *F*. Como se muestra en la tabla, *F* se define mediante varias operaciones aritméticas o booleanas sobre las entradas *A* y *B*. En esta tabla + significa suma aritmética y −, resta aritmética. Para evitar la confusión, la tabla usa las palabras XOR, OR y AND para las operaciones booleanas. Cada una de éstas se realizan en forma de bits. Por ejemplo, $F = A$ AND B produce el resultado de cuatro bits $f_0 = a_0b_0, f_1 = a_1b_1, f_2 = a_2b_2$ y $f_3 = a_3b_3$.

En la figura 6.48 se muestra cómo describir la funcionalidad de la ALU 74381 con código de VHDL. El paquete std_logic_unsigned, expuesto en la sección 5.5.4, se incluye para que las señales A y B de STD_LOGIC_VECTOR puedan usarse en operaciones aritméticas sin signo. La instrucción case mostrada corresponde directamente a la tabla 6.1. Para comprobar la funcionalidad del código, se sintetiza un circuito para implementarlo en un CPLD. En la figura 6.49 se ilustra un ejemplo de simulación de tiempo. Para cada combinación de *s*, el circuito genera la operación booleana o aritmética apropiada.

6.6.8 Operadores de VHDL

En esta sección estudiaremos los operadores de VHDL útiles para sintetizar circuitos lógicos. En la tabla 6.2 se enumeran en grupos que reflejan el tipo de operación realizada.

Para ilustrar los resultados que los diversos operadores producen emplearemos los vectores de tres bits A(2 DOWNTO 0), B(2 DOWNTO 0) y C(2 DOWNTO 0).

Operadores lógicos

Los operadores lógicos pueden utilizarse con los tipos de operandos bit o booleano. Los operandos pueden ser escalares de un solo bit o vectores multibit. Por ejemplo, la instrucción

```
C <= NOT A;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
    PORT ( s      : IN   STD_LOGIC_VECTOR(2 DOWNTO 0) ;
           A, B   : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           F      : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
        CASE s IS
            WHEN "000" =>
                F <= "0000" ;
            WHEN "001" =>
                F <= B - A ;
            WHEN "010" =>
                F <= A - B ;
            WHEN "011" =>
                F <= A + B ;
            WHEN "100" =>
                F <= A XOR B ;
            WHEN "101" =>
                F <= A OR B ;
            WHEN "110" =>
                F <= A AND B ;
            WHEN OTHERS =>
                F <= "1111" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 6.48 Código que representa la funcionalidad del chip de la ALU 74381.

produce el resultado $c_2 = \overline{a}_2, c_1 = \overline{a}_1$ y $c_0 = \overline{a}_0$, donde a_i y c_i son los bits de los vectores A y C.

La instrucción

```
C <= A AND B;
```

genera $c_2 = a_2 \cdot b_2, c_1 = a_1 \cdot b_1$ y $c_0 = a_0 \cdot b_0$. Los otros operadores conducen a evaluaciones similares.

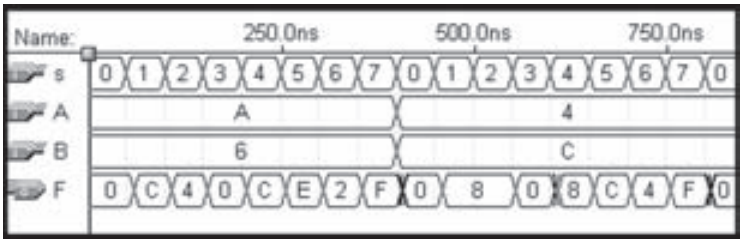


Figura 6.49 Simulación de tiempo para el código de la figura 6.48.

Tabla 6.2 Operadores de VHDL (usados para síntesis).

Categoría de operador	Símbolo de operador	Operación realizada
Lógica	AND	AND
	OR	OR
	NAND	Not AND
	NOR	Not OR
	XOR	XOR
	XNOR	Not XOR
	NOT	NOT
Relacional	=	Igualdad
	/=	Desigualdad
	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
Aritmética	+	Suma
	−	Resta
	*	Multiplicación
	/	División
Concatenación	&	Concatenación
Corrimiento y rotación	SLL	Corrimiento lógico izquierdo
	SRL	Corrimiento lógico derecho
	SLA	Corrimiento aritmético izquierdo
	SRA	Corrimiento aritmético derecho
	ROL	Rotación izquierda
	ROR	Rotación derecha

Operadores relacionales

Los operadores relacionales se emplean para comparar expresiones. El resultado de la comparación es TRUE (verdadero) o FALSE (falso). Las expresiones que se comparan deben ser del mismo tipo. Por ejemplo, si $A = 011$ y $B = 010$, entonces $A > B$ se evalúa TRUE, y B/= "010" FALSE.

Operadores aritméticos

En el capítulo 5 vimos los operadores aritméticos, que, naturalmente, realizan operaciones aritméticas comunes. Por tanto,

```
C <= A + B;
```

coloca en *C* la suma de tres bits de *A* más *B*, mientras que

```
C <= A - B;
```

coloca en *C* la diferencia de *A* y *B*. La operación

```
C <= -A;
```

coloca en *C* el complemento a 2 de *A*.

La mayor parte de las herramientas CAD de síntesis soporta las operaciones suma, resta y multiplicación. Sin embargo, casi nunca soporta la división. Cuando el compilador de VHDL encuentra un operador aritmético, lo sintetiza con un módulo apropiado de una biblioteca.

Operador de concatenación

Este operador concatena dos o más vectores para crear un vector más grande. Por ejemplo,

```
D <= A & B;
```

define el vector de seis bits $D = a_2a_1a_0b_2b_1b_0$. De manera similar, la concatenación

```
E <= "111" & A & "00";
```

produce el vector de ocho bits $E = 111a_2a_1a_000$.

Operadores de corrimiento y rotación

Un operando vector se puede correr a la derecha o a la izquierda un número de bits especificado como constante. Cuando los bits se corren, las posiciones de bit vacantes se rellenan con 0. Por ejemplo,

```
B <= A SLL 1;
```

resulta en $b_2 = a_1$, $b_1 = a_0$ y $b_0 = 0$. Del mismo modo,

```
B <= A SRL 2;
```

produce $b_2 = b_1 = 0$ y $b_0 = a_2$.

El corrimiento aritmético izquierdo, SLA, causa el mismo efecto que el SLL. Pero el corrimiento aritmético derecho, SRA, realiza la extensión del signo duplicando el bit del signo hacia las posiciones vacantes a la izquierda después del corrimiento. Por tanto

```
B <= A SRA 1;
```

produce $b_2 = a_2$, $b_1 = a_2$ y $b_0 = a_1$.

Un operando también se puede rotar, en cuyo caso los bits corridos de un extremo se colocan en las posiciones vacantes del otro. Por ejemplo,

```
B <= A ROR 2;
```

produce $b_2 = a_1$, $b_1 = a_0$ y $b_0 = a_2$.

Precedencia del operador

Los operadores de diferentes categorías tienen distintas precedencias. Los que pertenecen a la misma categoría tienen la misma precedencia y se evalúan de izquierda a derecha en una expresión. Es una buena práctica usar paréntesis para indicar el orden en que se desea evaluar una operación en la expresión. Para ilustrar este aspecto, considérese la instrucción

```
S <= A + B + C + D;
```

que define la suma de cuatro operandos vector. El compilador de VHDL sintetizará un circuito como si la expresión estuviese escrita en la forma $((A + B) + C) + D$, que produce una cascada de tres sumadores de modo que la suma final estará disponible después de un retraso de propagación a través de tres sumadores. Al escribir una instrucción como

```
S <= (A + B) + (C + D);
```

el circuito sintetizado todavía tendrá tres sumadores, pero como las sumas $A + B$ y $C + D$ se generan en paralelo, la suma final estará disponible después de un retraso de propagación a través de sólo dos sumadores.

En la tabla 6.2 se agrupan los operadores de manera informal de acuerdo con su funcionalidad. Se muestran sólo los empleados para sintetizar circuitos lógicos. VHDL estándar especifica operadores adicionales, útiles para propósitos de simulación y documentación. Todos los operadores están agrupados en diferentes clases, con un orden de precedencia definido entre ellas. Este tema se aborda en el Apéndice A, sección A.3.

6.7 COMENTARIOS FINALES

En este capítulo explicamos varios conceptos fundamentales de los circuitos. Los ejemplos que los usan para construir a partir de ahí circuitos más grandes se presentarán en los capítulos 7 y 10. Para describir eficientemente los circuitos fundamentales hemos expuesto varios constructores de VHDL. En muchos casos un circuito puede describirse de varias formas, con diferentes constructores. Un circuito que puede describirse con una asignación de señal seleccionada también puede ser descrito mediante una instrucción case. Los circuitos que encajan bien con las asignaciones de señal condicional también son adecuados para instrucciones if-then-else. En general, no hay reglas claras que dicten cuándo debe preferirse un tipo de instrucción de asignación en vez de otro. Con la experiencia el usuario desarrolla cierto sentido acerca de cuáles son los tipos de instrucciones que funcionan bien en una situación de diseño concreta. La preferencia personal también influye en cómo se escribe el código.

VHDL no es un lenguaje de programación, y su código no debe escribirse como si se estuviese desarrollando un programa para computadora. Las instrucciones de asignación concurrente y secuencial expuestas en el capítulo sirven para crear circuitos grandes y complejos. Una buena forma de diseñar tales circuitos consiste en construirlos con módulos bien definidos, como lo ejemplificamos para los multiplexores, decodificadores, codificadores, etc. En los capítulos 7 y 8 se ofrecen ejemplos adicionales que utilizan las instrucciones de VHDL presentadas en este capítulo. En el capítulo 10 veremos varios ejemplos de uso de código de VHDL para describir sistemas digitales más grandes. Para más información acerca de VHDL, el lector puede consultar obras más especializadas [5-10].

En el capítulo siguiente presentaremos los circuitos lógicos que tienen la capacidad de almacenar valores de señal lógica en elementos de memoria.

6.8 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección presentamos algunos problemas típicos que puede encontrar el lector, al tiempo que se muestra cómo resolverlos.

Ejemplo 6.25 **Problema:** Implemente la función $f(w_1, w_2, w_3) = \sum m(0, 1, 3, 4, 6, 7)$ con un decodificador binario tres a ocho y una compuerta OR.

Solución: El decodificador genera una salida por separado para cada mintérmino de la función requerida. Entonces esas salidas se combinan en la compuerta OR, lo que produce el circuito de la figura 6.50.

Ejemplo 6.26 **Problema:** Derive un circuito que implemente un codificador binario ocho a tres.

Solución: La tabla de verdad del codificador se muestra en la figura 6.51. Sólo se presentan las columnas para las que una variable de entrada sola es igual a 1; las demás filas pueden tratarse como casos no importa. A partir de la tabla de verdad se advierte que el circuito deseado se define mediante las ecuaciones

$$y_2 = w_4 + w_5 + w_6 + w_7$$
$$y_1 = w_2 + w_3 + w_6 + w_7$$
$$y_0 = w_1 + w_3 + w_5 + w_7$$

Ejemplo 6.27 **Problema:** Implemente la función

$$f(w_1, w_2, w_3, w_4) = \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5$$

con un multiplexor cuatro a uno y las menos compuertas posibles. Suponga que sólo están disponibles las entradas sin complementar w_1, w_2, w_3 y w_4.

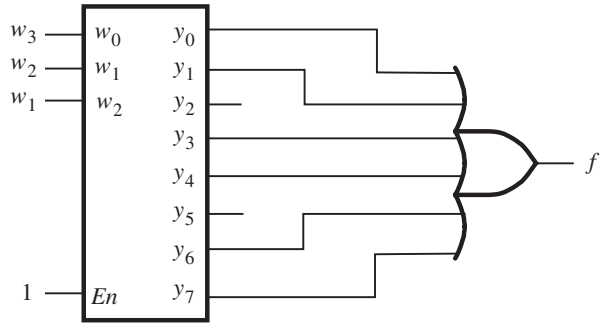


Figura 6.50 Circuito para el ejemplo 6.25.

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figura 6.51 Tabla de verdad para un codificador binario ocho a tres.

Solución: Puesto que las variables w_1 y w_4 aparecen en más términos producto que las otras tres variables en la expresión para f, realice expansión de Shannon con respecto a esas dos variables. La expansión produce

$$\begin{aligned} f &= \overline{w}_1\overline{w}_4 f_{\overline{w}_1\overline{w}_4} + \overline{w}_1 w_4 f_{\overline{w}_1 w_4} + w_1\overline{w}_4 f_{w_1\overline{w}_4} + w_1 w_4 f_{w_1 w_4} \\ &= \overline{w}_1\overline{w}_4(\overline{w}_2\overline{w}_5) + \overline{w}_1 w_4(w_3 w_5) + w_1\overline{w}_4(w_2 + w_3) + w_1 w_2(1) \end{aligned}$$

Podemos usar una compuerta NOR para implementar $\overline{w}_2\overline{w}_5 = \overline{w_2 + w_5}$. También necesitamos dos compuertas, una AND y otra OR. El circuito completo se presenta en la figura 6.52.

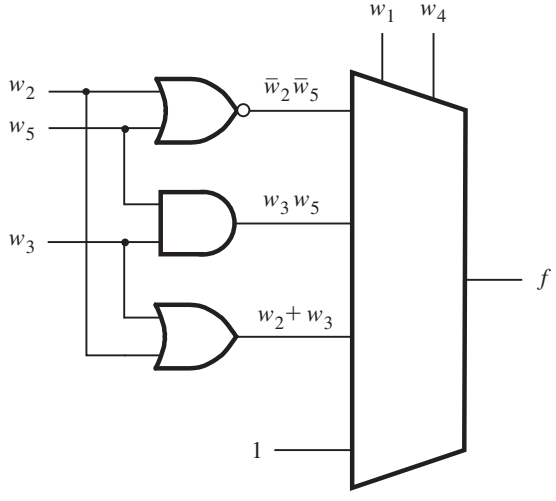


Figura 6.52 Circuito para el ejemplo 6.27.

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Figura 6.53 Conversión de código binario a Gray.

Ejemplo 6.28 **Problema:** En el capítulo 4 señalamos que las filas y columnas de un mapa de Karnaugh se etiquetan con código Gray, que es un código en el que las combinaciones consecutivas difieren sólo en una variable. En la figura 6.53 se muestra la conversión de código binario a Gray de tres bits. Diseñe un circuito que pueda convertir un código binario a Gray de acuerdo con la figura.

Solución: A partir de la figura se sigue que

$$\begin{aligned} g_2 &= b_2 \\ g_1 &= b_1\overline{b}_2 + \overline{b}_1 b_2 \\ &= b_1 \oplus b_2 \\ g_0 &= b_0\overline{b}_1 + \overline{b}_0 b_1 \\ &= b_0 \oplus b_1 \end{aligned}$$

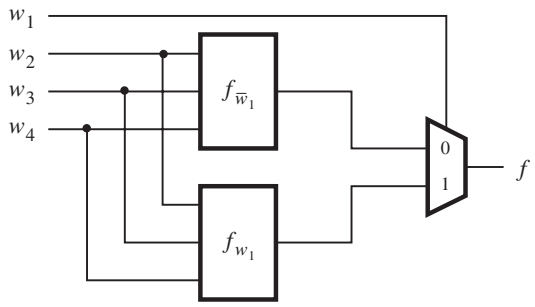
Ejemplo 6.29 **Problema:** En la sección 6.1.2 mostramos que cualquier función lógica puede descomponerse aplicando el teorema de expansión de Shannon. Para una función de cuatro variables, $f(w_1, \ldots, w_4)$, la expansión respecto a w_1 es

$$f(w_1, \ldots, w_4) = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

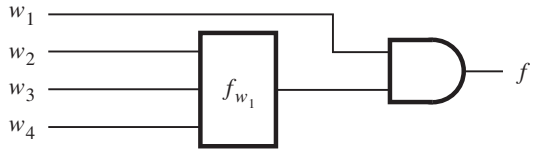
En la figura 6.54*a* se muestra un circuito que implementa esta expresión.
a) Si la descomposición produce $f_{\overline{w}_1} = 0$, entonces el multiplexor de la figura puede sustituirse con una sola compuerta lógica. Muestre este circuito.
b) Repita el inciso *a*) para el caso donde $f_{w_1} = 1$.

Solución: Los circuitos deseados se muestran en los incisos *b*) y *c*) de la figura 6.54.

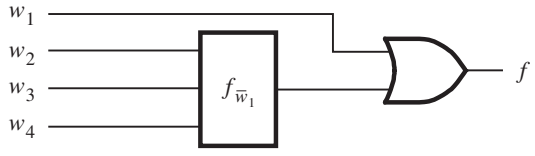
Ejemplo 6.30 **Problema:** En varios FPGA comerciales, los bloques lógicos son LUT 4. ¿Cuál es el mínimo número de LUT 4 necesarios para construir un multiplexor cuatro a uno con entradas de selección s_1 y s_0 y entradas de datos w_3, w_2, w_1 y w_0?



a) Expansión de Shannon de la función f



b) Solución para el inciso *a*)



c) Solución para el inciso *b*)

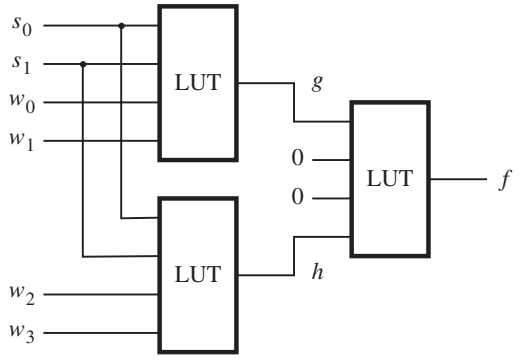
Figura 6.54 Circuitos para el ejemplo 6.29.

Solución: Un intento directo es usar simplemente la expresión que define al multiplexor cuatro a uno

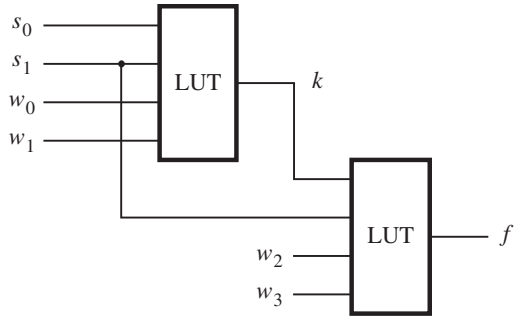
$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

Sean $g = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1$ y $h = s_1\overline{s}_0w_2 + s_1s_0w_3$, de modo que $f = g + h$. Esta descomposición conduce al circuito de la figura 6.55*a*, que requiere tres LUT.

Cuando se diseñan circuitos lógicos, uno a veces topa con una idea ingeniosa que desemboca en una implementación superior. En la figura 6.55*b* se muestra cómo implementar el multiplexor con sólo dos LUT, con base en la observación siguiente. La tabla de verdad de la figura 6.2*b* indica que cuando $s_1 = 0$, la salida debe ser w_0 o w_1, como lo determine el valor de s_0. Esto puede generarse mediante la primera LUT. La segunda LUT debe realizar la elección entre w_2 y w_3 cuando $s_1 = 1$. Pero la elección se puede hacer sólo al conocer el valor de s_0. En virtud de que resulta imposible tener cinco entradas en la LUT, es preciso pasar más información de



a) Con tres LUT



b) Con dos LUT

Figura 6.55 Circuitos para el ejemplo 6.30.

la primera a la segunda LUT. Observe que cuando $s_1 = 1$, la salida f será igual a w_2 o a w_3, en cuyo caso no es necesario conocer los valores de w_0 y w_1. Por tanto, en este caso puede pasarse el valor de s_0 a través de la primera LUT, en lugar de w_0 o w_1. Esto puede efectuarse haciendo la función de esta LUT

$$k = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1s_0$$

Entonces, la segunda LUT realiza la función

$$f = \overline{s}_1k + s_1\overline{k}w_3 + s_1kw_4$$

Ejemplo 6.31 **Problema:** En los sistemas digitales a menudo es necesario tener circuitos que puedan correr los bits de un vector una o más posiciones de bit a izquierda o derecha. Diseñe un circuito que pueda

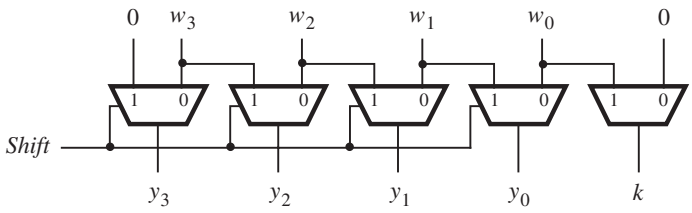


Figura 6.56 Circuito de corrimiento.

correr un vector de cuatro bits $W = w_3w_2w_1w_0$ una posición de bit a la derecha cuando una señal de control *Shift* es igual a 1. Sean las salidas del circuito un vector de cuatro bits $Y = y_3y_2y_1y_0$ y una señal k, tal que si *Shift* = 1, entonces $y_3 = 0$, $y_2 = w_3$, $y_1 = w_2$, $y_0 = w_1$, y $k = w_0$. Si *Shift* = 0, entonces $Y = W$ y $k = 0$.

Solución: El circuito requerido puede implementarse con cinco multiplexores dos a uno como se muestra en la figura 6.56. La señal *Shift* se usa como la entrada de selección a cada multiplexor.

Ejemplo 6.32

Problema: El circuito de corrimiento del ejemplo 6.31 corre los bits de un vector de entrada una posición de bit a la derecha. Rellena el bit vacante al lado izquierdo con 0. Un circuito de corrimiento más flexible puede correr más posiciones de bit a la vez. Si los bits que se corren se colocan en las posiciones vacantes a la izquierda, entonces el circuito efectivamente rota los bits del vector entrada un número específico de posiciones de bit. Tal circuito con frecuencia se llama *registro de corrimiento en cilindro*. Diseñe un registro de corrimiento en cilindro que rote los bits 0, 1, 2 o 3 posiciones de bit según determine la combinación de dos señales de control s_1 y s_0.

Solución: La acción requerida se muestra en la figura 6.57*a*. El registro de corrimiento en cilindro puede implementarse con cuatro multiplexores cuatro a uno, como se muestra en la figura 6.57*b*. Las señales de control s_1 y s_0 se usan como las entradas de selección a los multiplexores.

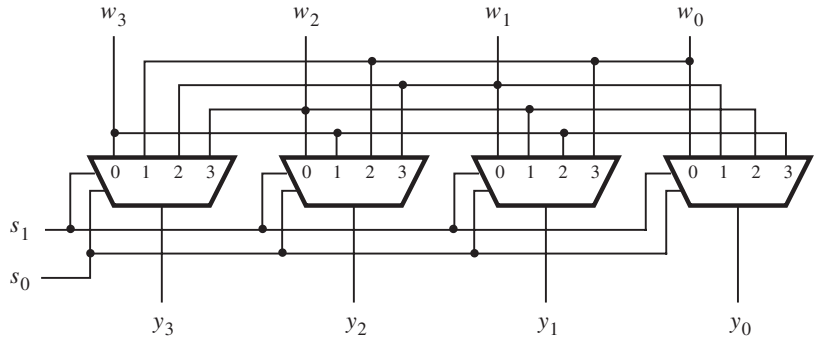
Ejemplo 6.33

Problema: Escriba código de VHDL que represente el circuito de la figura 6.19. Utilice en su código la entidad *dec2to4* de la figura 6.30 como un subcircuito.

Solución: El código se muestra en la figura 6.58. Note que la entidad *dec2to4* puede incluirse en el mismo archivo, como hemos hecho en la figura, pero también puede estar en otro archivo en el directorio de proyecto.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

a) Tabla de verdad



b) Circuito

Figura 6.57 Circuito de registro de corrimiento en cilindro.

Ejemplo 6.34 **Problema:** Escriba código de VHDL que represente el circuito de corrimiento de la figura 6.56.

Solución: Existen dos enfoques posibles: estructural y por comportamiento. En la figura 6.59 se ofrece una descripción estructural. El constructor IF se usa para definir el corrimiento deseado de los bits individuales. Un compilador de VHDL típico implementará este código con multiplexores dos a uno, como se muestra en la figura 6.56.

En la figura 6.60 se presenta una especificación por comportamiento. Utiliza el operador de corrimiento SRL. Puesto que la biblioteca *ieee.numeric_std.all* soporta los operadores de corrimiento y rotación, debe incluirse en el código. Note que los vectores *w* y *y* se definen como tipo UNSIGNED.

Ejemplo 6.35 **Problema:** Escriba código de VHDL que defina el registro de corrimiento en cilindro de la figura 6.57.

Solución: La forma más sencilla de especificar el corrimiento en cilindro es con el operador de rotación de VHDL. El código completo se presenta en la figura 6.61.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT ( s   : IN    STD_LOGIC_VECTOR( 1 DOWNTO 0 ) ;
           w   : IN    STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
           f   : OUT   STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Structure OF mux4to1 IS
    COMPONENT dec2to4
        PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En  : IN    STD_LOGIC ;
               y   : OUT   STD_LOGIC_VECTOR(0 TO 3) );
    END COMPONENT;
    SIGNAL High : STD_LOGIC ;
    SIGNAL y : STD_LOGIC_VECTOR( 3 DOWNTO 0 ) ;
BEGIN
    decoder: dec2to4 PORT MAP ( s, '1', y ) ;
    f <= (w(0) AND y(0)) OR (w(1) AND y(1)) OR
         (w(2) AND y(2)) OR w(3) AND y(3) ) ;
END Structure ;


LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw<= En & w ;
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;
END Behavior ;
```

Figura 6.58 Código de VHDL para el ejemplo 6.38.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shifter IS
    PORT ( w      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Shift  : IN  STD_LOGIC ;
           y      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           k      : OUT STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = '1' THEN
            y(3) <= '0' ;
            y(2 DOWNTO 0) <= w(3 DOWNTO 1) ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= '0' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.59 Código estructural de VHDL que especifica el circuito de corrimiento de la figura 6.56.

PROBLEMAS

Al final del libro se proporcionan las respuestas de los problemas marcados con asterisco.

6.1 Muestre cómo implementar la función $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$ con un decodificador binario tres a ocho y una compuerta OR.

6.2 Muestre cómo implementar la función $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$ con un decodificador binario tres a ocho y una compuerta OR.

***6.3** Considere la función $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. Use la tabla de verdad a fin de derivar un circuito para f que use un multiplexor dos a uno.

6.4 Repita el problema 6.3 para la función $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

***6.5** Para la función $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$, use la expansión de Shannon para derivar una implementación que utilice un multiplexor dos a uno y cualesquiera otras compuertas necesarias.

6.6 Repita el problema 6.5 para la función $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY shifter IS
    PORT ( w     : IN   UNSIGNED(3 DOWNTO 0) ;
           Shift : IN   STD_LOGIC ;
           y     : OUT  UNSIGNED(3 DOWNTO 0) ;
           k     : OUT  STD_LOGIC ) ;
END shifter ;

ARCHITECTURE Behavior OF shifter IS
BEGIN
    PROCESS (Shift, w)
    BEGIN
        IF Shift = "1" THEN
            y <= w SRL 1 ;
            k <= w(0) ;
        ELSE
            y <= w ;
            k <= "0" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 6.60 Código por comportamiento de VHDL que especifica el circuito de corrimiento de la figura 6.56.

6.7 Considere la función $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1w_3$. Muestre cómo se puede usar la aplicación repetida de la expansión de Shannon para derivar los mintérminos de f.

6.8 Repita el problema 6.7 para $f = w_2 + \overline{w}_1\overline{w}_3$.

6.9 Demuestre el teorema de expansión de Shannon expuesto en la sección 6.1.2.

***6.10** En la sección 6.1.2 se muestra la expansión de Shannon en forma de suma de productos. Con el principio de dualidad, derive la expresión equivalente en forma de producto de sumas.

6.11 Considere la función $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1w_2w_3$. Proporcione un circuito que implemente f con el mínimo número de LUT de dos entradas. Muestre la tabla de verdad implementada dentro de cada LUT.

***6.12** Para la función del problema 6.11, el costo de la expresión en mínima suma de productos es 14, que incluye cuatro compuertas y 10 entradas a las compuertas. Utilice la expansión de Shannon para derivar un circuito multinivel que tenga un costo menor y proporcione el costo de su circuito.

6.13 Considere la función $f(w_1, w_2, w_3, w_4) = \sum m(0, 1, 3, 6, 8, 9, 14, 15)$. Derive una implementación que use el mínimo número posible de LUT de tres entradas.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;

ENTITY barrel IS
    PORT ( w  : IN   UNSIGNED(3 DOWNTO 0) ;
           s  : IN   UNSIGNED(1 DOWNTO 0) ) ;
           y  : OUT  UNSIGNED(3 DOWNTO 0) ) ;
END barrel ;

ARCHITECTURE Behavior OF barrel IS
BEGIN
    PROCESS (s, w)
    BEGIN
        CASE s IS
            WHEN "00" =>
                y <= w ;
            WHEN "01" =>
                y <= w ROR 1 ;
            WHEN "10" =>
                y <= w ROR 2 ;
            WHEN OTHERS =>
                y <= w ROR 3 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 6.61 Código de VHDL que especifica el circuito de corrimiento en cilindro de la figura 6.57.

***6.14** Dé dos ejemplos de funciones lógicas con cinco entradas, $w_1, \ldots, w_5$, que se pueda realizar con dos LUT de cuatro entradas.

6.15 Para la función *f* del ejemplo 6.27, realice expansión de Shannon respecto de las variables w_1 y w_2, en lugar de w_1 y w_4. ¿Cómo se compara el circuito resultante con el de la figura 6.52?

6.16 Actel Corporation fabrica una familia de FPGA llamada Act 1, que tiene el bloque lógico basado en multiplexor que se muestra en la figura P6.1. Indique cómo implementar la función $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ sólo con un bloque lógico Act 1.

6.17 Muestre cómo se realiza la función $f = w_1\overline{w}_3 + \overline{w}_1w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ con bloques lógicos Act 1. Note que no hay compuertas NOT en el chip; por tanto, los complementos de las señales deben generarse con multiplexores en el bloque lógico.

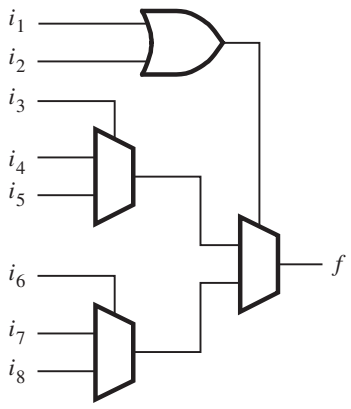


Figura P6.1 Bloque lógico Act 1 de Actel.

***6.18** Considere el código de VHDL de la figura P6.2. ¿Qué tipo de circuito representa? Comente si el estilo de código usado es una buena elección para el circuito que representa.

6.19 Escriba el código de VHDL que represente la función del problema 6.1, empleando una asignación de señal seleccionada.

6.20 Escriba el código de VHDL que represente la función del problema 6.2, usando una asignación de señal seleccionada.

6.21 Con una asignación de señal seleccionada, escriba el código de VHDL para un codificador binario cuatro a dos.

6.22 Con una asignación de señal condicional, escriba el código de VHDL para un codificador binario ocho a tres.

6.23 Derive el circuito para un codificador de prioridad ocho a tres.

6.24 Con una asignación de señal condicional, escriba el código de VHDL para un codificador de prioridad ocho a tres.

6.25 Repita el problema 6.24 ahora con una instrucción if-then-else.

6.26 Cree una entidad de VHDL llamada *if2to4* que represente un decodificador binario que use una instrucción if-then-else. Cree una segunda entidad llamada *h3to8* que represente el decodificador binario tres a ocho de la figura 6.17, con dos instancias de la entidad *if2to4*.

6.27 Cree una entidad de VHDL llamada *h6to64* que represente un decodificador binario 6 a 64. Use la estructura arborescente de la figura 6.18 en la que el decodificador 6 a 64 se construya con cinco instancias del decodificador *h3to8* creado en el problema 6.26.

6.28 Escriba el código de VHDL para un convertidor de código BCD a siete segmentos usando una asignación de señal seleccionada.

***6.29** Derive expresiones en mínima suma de productos para las salidas *a*, *b* y *c* de la pantalla de siete segmentos de la figura 6.25.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( w              : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En             : IN    STD_LOGIC ;
           y0, y1, y2, y3 : OUT   STD_LOGIC ) ;
END problem ;

ARCHITECTURE Behavior OF problem IS
BEGIN
    PROCESS (w, En)
    BEGIN
        y0 <= '0' ; y1 <= '0' ; y2 <= '0' ; y3 <= '0' ;
        IF En    '1' THEN
            IF w = "00" THEN y0 <= '1' ;
            ELSIF w = "01" THEN y1 <= '1' ;
            ELSIF w = "10" THEN y2 <= '1' ;
            ELSE y3 <= '1' ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura P6.2 Código para el problema 6.18.

6.30 Derive expresiones en mínima suma de productos para las salidas *d, e, f* y *g* de la pantalla de siete segmentos de la figura 6.25.

6.31 Diseñe un circuito de corrimiento, similar al de la figura 6.56, que pueda correr un vector entrada de cuatro bits, $W = w_3w_2w_1w_0$, una posición de bit a la derecha cuando la señal de control *Right* sea igual a 1, y una posición de bit a la izquierda cuando la señal de control *Left* sea igual a 1. Cuando *Right* = *Left* = 0, la salida del circuito debe ser la misma que el vector entrada. Suponga que la condición *Right* = *Left* = 1 nunca ocurrirá.

6.32 En la figura 6.21 se muestra un diagrama de bloques de una ROM. Un circuito que implementa una pequeña ROM, con cuatro hileras y cuatro columnas, se muestra en la figura P6.3. Cada X en la figura representa un interruptor que determina si la ROM produce un 1 o un 0 cuando dicha ubicación se lee.

a) Muestre cómo realizar un interruptor (X) empleando un solo transistor NMOS.

b) Dibuje completo el circuito ROM de 4 × 4 utilizando los interruptores creados en el inciso anterior. La ROM debe programarse para almacenar los bits 0101 en la fila 0 (la superior), 1010 en la 1, 1100 en la 2 y 0011 en la 3 (la fila inferior).

c) Muestre cómo implementar cada interruptor (X) como un interruptor programable (en oposición a ofrecer 1 o 0 de manera permanente), con una celda EEPROM, como se muestra en la figura 3.64. Describa brevemente cómo se usa la celda de almacenamiento.

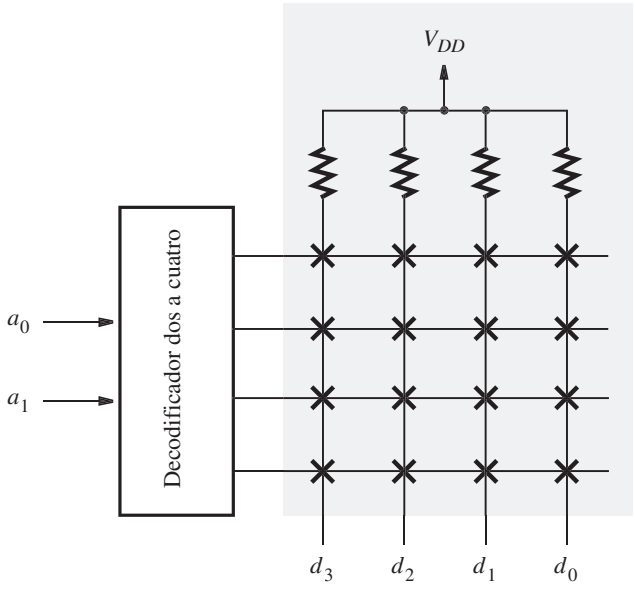


Figura P6.3 Circuito ROM de 4 × 4.

6.33 Muestre el circuito completo para una ROM que use las celdas de almacenamiento diseñadas en el inciso *a*) del problema 6.32 que realice las funciones lógicas

$$d_3 = a_0 \oplus a_1$$
$$d_2 = \overline{a_0 \oplus a_1}$$
$$d_1 = a_0 a_1$$
$$d_0 = a_0 + a_1$$

BIBLIOGRAFÍA

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits", *Transactions AIEE* 57 (1938), pp. 713-723.
2. Actel Corporation, "MX FPGA Data Sheet", http://www.actel.com
3. QuickLogic Corporation, "pASIC 3 FPGA Data Sheet", http://www.quicklogic.com

4. R. Landers, S. Mahant-Shetti y C. Lemonds, "A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays", *IEEE Journal of Solid-State Circuits* 30, núm. 4, (abril de 1995).
5. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2a. ed. (McGraw-Hill: Nueva York, 1998).
6. J. Bhasker, *A VHDL Primer*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
7. D. L. Perry, *VHDL*, 3a. ed. (McGraw-Hill: Nueva York, 1998).
8. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
9. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).
10. D. J. Smith, *HDL Chip Design* (Doone Publications: Madison, AL, 1996).

capítulo

7

FLIP-FLOPS, REGISTROS, CONTADORES Y UN PROCESADOR SIMPLE

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

- Circuitos lógicos que pueden almacenar información
- Flip-flops, que almacenan un solo bit
- Registros, que almacenan varios bits
- Registros de corrimiento, que desplazan el contenido del registro
- Contadores de diversos tipos
- Constructores de VHDL usados para implementar elementos de almacenamiento
- Diseño de subsistemas pequeños

En capítulos anteriores consideramos los circuitos combinacionales en los que el valor de cada salida depende exclusivamente de los valores de las señales aplicadas a las entradas. Hay otro tipo de circuitos lógicos en los que los valores de la salida no sólo dependen de los valores presentes de las entradas, sino también del comportamiento previo del circuito. Estos circuitos incluyen elementos de almacenamiento que guardan los valores de las señales lógicas. Se dice que el contenido de los elementos de almacenamiento representa el *estado* del circuito. Cuando las entradas del circuito cambian de valor, los nuevos valores de entrada dejan el circuito en el mismo estado o hacen que éste entre en un estado nuevo. Con el tiempo el circuito pasa por una secuencia de estados como resultado de los cambios en las entradas. Los circuitos que se comportan de esta manera se conocen como *circuitos secuenciales*.

En este capítulo estudiaremos los circuitos que se utilizan como elementos de almacenamiento, pero antes, por medio de un ejemplo simple, explicaremos por qué son necesarios. Supóngase que deseamos controlar un sistema de alarma como se muestra en la figura 7.1. El mecanismo de alarma responde a la entrada de control $On/\overline{Off}$. Está encendido cuando $On/\overline{Off} = 1$, y está pagado cuando $On/\overline{Off} = 0$. La operación que se busca es que la alarma se active cuando el sensor genere una señal de voltaje positivo, *Set*, en respuesta a algún hecho no deseado. Una vez que la alarma se activa debe permanecer en tal estado aun si la salida del sensor regresa a 0. La alarma se apaga manualmente por medio de una entrada *Reset*. El circuito requiere un elemento de memoria para recordar que la alarma ha de estar activa hasta que la señal *Reset* llegue.

En la figura 7.2 se muestra un elemento de memoria rudimentario que consta de un ciclo con dos inversores. Si suponemos que $A = 0$, entonces $B = 1$. El circuito mantendrá estos valores indefinidamente. Decimos entonces que el circuito se halla en el *estado* definido por ellos. Si suponemos que $A = 1$, entonces $B = 0$, y el circuito mantendrá este estado de manera indefinida. Por tanto, el circuito tiene dos estados posibles. Este circuito no es útil, ya que carece de un medio práctico para cambiar de estado.

En la figura 7.3 se muestra un circuito más útil. Incluye un mecanismo para cambiar el estado del circuito de la figura 7.2 mediante dos compuertas de transmisión del tipo estudiado en la sección 3.9. Una de las compuertas, *TG*1, se usa para conectar la terminal de entrada *Data* con el

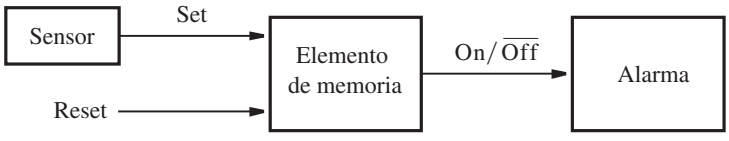


Figura 7.1 Control de un sistema de alarma.

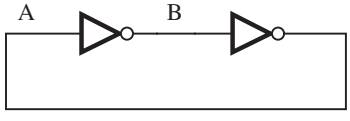


Figura 7.2 Un elemento de memoria simple.

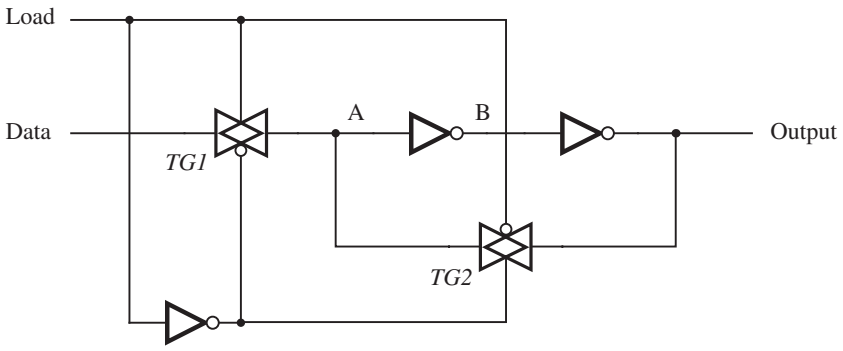


Figura 7.3 Un elemento de memoria controlado.

punto *A* del circuito. La otra compuerta, *TG*2, sirve como interruptor en el *lazo de retroalimentación* que mantiene el estado del circuito. Las compuertas de transmisión están controladas por la señal *Load*. Si *Load* = 1, entonces *TG*1 está abierta y el punto *A* tendrá el mismo valor que la entrada *Data*. Como el valor almacenado actualmente en *Output* tal vez no sea el mismo que *Data*, el lazo de retroalimentación se interrumpe al hacer que *TG*2 se cierre cuando *Load* = 1. Cuando *Load* = 0, entonces *TG*1 se cierra y *TG*2 se abre. La ruta de retroalimentación se cierra y el elemento de memoria conservará su estado siempre que *Load* = 0. Este elemento de memoria no puede aplicarse de modo directo al sistema de la figura 7.1, pero es útil para muchas otras aplicaciones, como veremos más adelante.

7.1 EL LATCH BÁSICO

En vez de usar las compuertas de transmisión podemos construir un circuito similar empleando compuertas lógicas ordinarias. En la figura 7.4 se presenta un elemento de memoria construido con compuertas NOR. Sus entradas, *Set* y *Reset*, proveen los medios para cambiar el estado, Q, del circuito. Una forma más común de dibujar éste se muestra en la figura 7.5*a*, donde se dice que las dos compuertas NOR están conectadas en un estilo de acoplamiento cruzado. El circuito se conoce como *latch básico* y su comportamiento se describe en la tabla de la figura 7.5*b*. Cuando las dos entradas, *R* y *S*, son iguales a cero, el latch mantiene su estado actual, que puede ser $Q_a = 0$ y $Q_b = 1$, o $Q_a = 1$ y $Q_b = 0$, lo cual se indica en la tabla señalando que las salidas Q_a y Q_b tienen valores

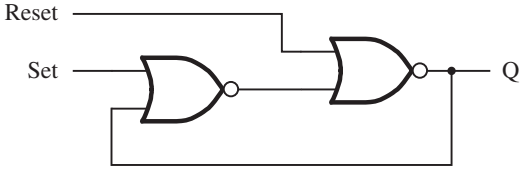
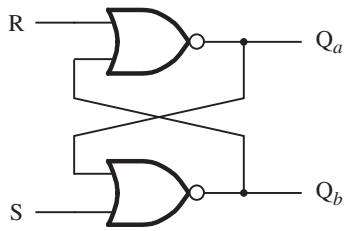


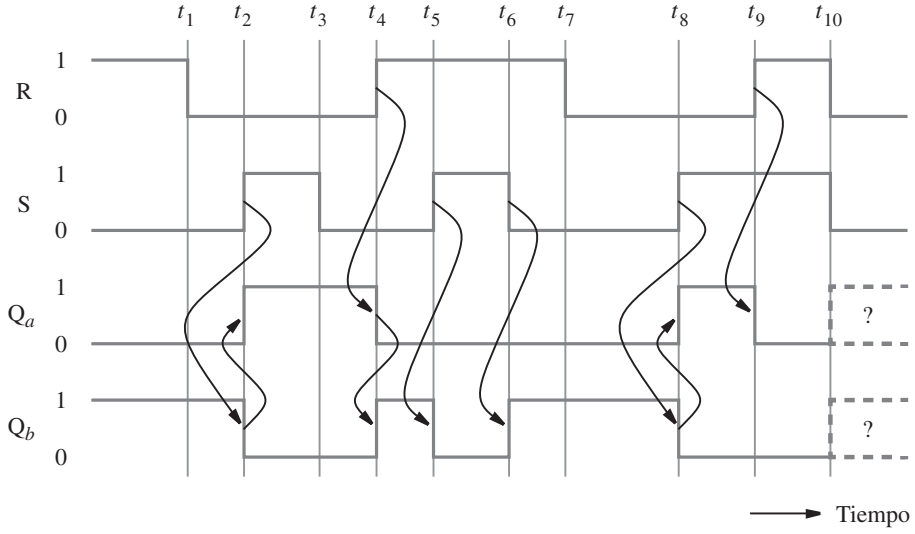
Figura 7.4 Un elemento de memoria con compuertas NOR.



a) Circuito

S	R	Q_a	Q_b	
0	0	0/1	1/0	(sin cambio)
0	1	0	1	
1	0	1	0	
1	1	0	0	

b) Tabla característica



c) Diagrama de tiempo

Figura 7.5 Un latch básico construido con compuertas NOR.

0/1 y 1/0, respectivamente. Obsérvese que en este caso Q_a y Q_b se complementan mutuamente. Cuando $R = 0$ y $S = 1$, el latch se *fija* en un estado donde $Q_a = 1$ y $Q_b = 0$ (estado conocido como *set*). Cuando $R = 1$ y $S = 0$, el latch se regresa a un estado en el que $Q_a = 0$ y $Q_b = 1$ (estado conocido como *reset*). La cuarta posibilidad es tener $R = S = 1$. En este caso, tanto Q_a como Q_b serán 0. La tabla de la figura 7.5*b* se parece a una tabla de verdad. Sin embargo, como no representa un circuito combinacional donde los valores de las salidas se determinan exclusivamente por los valores presentes en las entradas, recibe el nombre de *tabla característica* en vez de tabla de verdad.

En la figura 7.5*c* se presenta un diagrama de tiempo para el latch, suponiendo que el retraso de propagación a través de las compuertas NOR es insignificante. Desde luego, en un circuito de verdad los cambios en la forma de la onda estarán retrasados de acuerdo con los retrasos de propagación de las compuertas. Supóngase que inicialmente $Q_a = 0$ y $Q_b = 1$. El estado del latch permanece sin cambios hasta el tiempo t_2, cuando S se vuelve igual a 1, lo que ocasiona

que Q_b cambie a 0, lo cual a su vez hace que Q_a cambie a 1. La relación de causalidad se indica en el diagrama por medio de flechas. Cuando S pasa a 0 en t_3 no hay cambio en el estado porque tanto S como R son iguales a 0. En t_4 tenemos $R = 1$, lo que ocasiona que Q_a se establezca en 0, lo cual a su vez hace que Q_b pase a 1. En t_5 tanto S como R son iguales a 1, lo que causa que tanto Q_a como Q_b sean iguales a 0. Cuando S regresa a 0 en t_6 Q_b se vuelve igual a 1 de nuevo. En t_8 tenemos $S = 1$ y $R = 0$, lo que produce que $Q_b = 0$ y $Q_a = 1$. Una situación interesante ocurre en t_{10}. De t_9 a t_{10} tenemos $Q_a = Q_b = 0$ porque $R = S = 1$. Ahora, si tanto R como S cambian a 0 en t_{10}, entonces Q_a y Q_b se establecerán en 1. Pero tener Q_a y Q_b iguales a 1 forzará de inmediato que $Q_a = Q_b = 0$. Habrá una oscilación entre $Q_a = Q_b = 0$ y $Q_a = Q_b = 1$. Si los retrasos a través de las dos compuertas NOR son exactamente iguales, la oscilación continuará de manera indefinida. En un circuito real siempre habrá una diferencia en los retrasos por esas compuertas y con el tiempo el latch se fijará en uno de sus dos estados estables, pero no sabemos en cuál de ellos. Esta incertidumbre se indica con formas de ondas en líneas punteadas.

Las oscilaciones analizadas en el párrafo anterior ilustran que si bien el latch básico es un circuito simple, debe realizarse un análisis detallado para advertir por completo este comportamiento. En general, todo circuito que contenga una o más rutas de retroalimentación, tal que el estado del circuito dependa de los retrasos de propagación a través de compuertas lógicas, ha de diseñarse con cuidado. Comentaremos de manera pormenorizada las cuestiones relativas al tiempo en el capítulo 9.

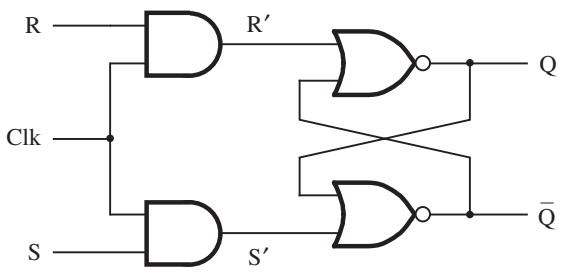
El latch de la figura 7.5*a* puede realizar las funciones necesarias para el elemento de memoria de la figura 7.1 al conectar la señal *Set* a la entrada S y *Reset* a la R. La salida Q_a proporciona la señal $On/\overline{Off}$ buscada. Para inicializar el funcionamiento del sistema de alarma, el latch se vuelve a establecer en 0 (reset). De esta forma la alarma se apaga. Cuando el sensor genera el valor lógico 1, el latch se establece en 1 y Q_a se vuelve igual a 1. Esto activa el mecanismo de alarma. Si la salida del sensor vuelve a 0, el latch conserva su estado donde $Q_a = 1$; por tanto, la alarma permanece activa. El único modo de apagarla es al reinicializar el latch, lo que se logra haciendo que la entrada *Reset* sea igual a 1.

7.2 LATCH SR ASÍNCRONO

En la sección 7.1 vimos que el latch SR básico puede servir como un elemento de memoria útil, ya que recuerda su estado cuando las dos entradas S y R son 0. Modifica su estado en respuesta a los cambios en las señales de estas entradas. Los cambios de estado ocurren en el momento en que suceden los cambios en las señales. Si no podemos controlar cuándo ocurren tales cambios, entonces no sabemos cuándo el latch puede cambiar de estado.

En el sistema de alarma de la figura 7.1, tal vez se quiera habilitarlo o inhabilitarlo todo por medio de una entrada de control, *Enable*. Así, cuando el sistema esté habilitado funcionará como se describió antes. En el modo inhabilitado, el cambiar la entrada *Set* de 0 a 1 no ocasionaría la activación de la alarma. El latch de la figura 7.5*a* no puede brindar la operación buscada, pero su circuito puede modificarse para responder a las señales de entrada S y R sólo cuando *Enable* = 1; de lo contrario, mantendrá su estado.

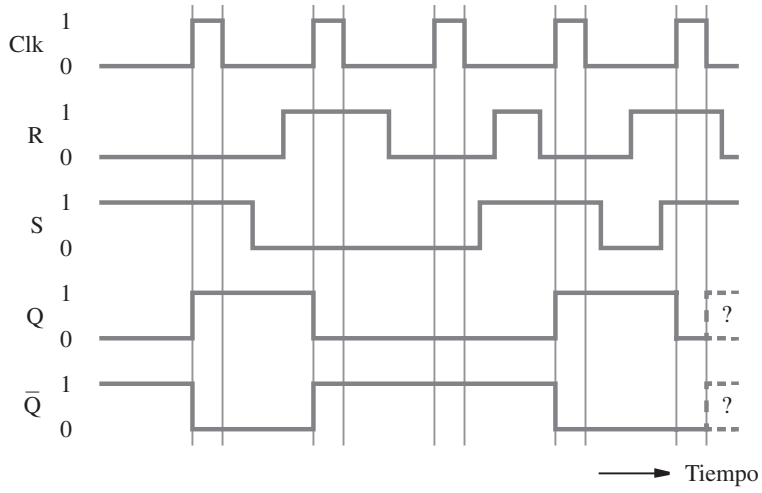
El circuito modificado se presenta en la figura 7.6*a*. Incluye dos compuertas AND que proporcionan el control descrito. Cuando la señal de control *Clk* es igual a 0, las entradas al latch S' y R' serán 0, independientemente de los valores de las señales S y R. Por consiguiente, el latch mantendrá



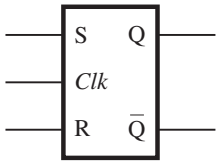
a) Circuito

Clk	S	R	Q(*t* + 1)
0	x	x	Q(*t*) (sin cambio)
1	0	0	Q(*t*) (sin cambio)
1	0	1	0
1	1	0	1
1	1	1	x

b) Tabla característica



c) Diagrama de tiempo



d) Símbolo gráfico

Figura 7.6 Latch SR asíncrono.

su estado actual siempre que $Clk = 0$. Cuando *Clk* cambia a 1, las señales S' y R' serán iguales que las señales *S* y *R*, respectivamente. Por tanto, en este modo el latch se comportará como describimos en la sección 7.1. Obsérvese que hemos usado el nombre *Clk* para la señal de control que permite que el latch se establezca en 1 o se inicialice en 0, en vez de llamarla señal *Enable*. La razón es que estos circuitos se usan en los sistemas digitales donde se busca permitir que los cambios de

estado de los elementos de memoria ocurran sólo en intervalos de tiempo bien definidos, como si estuvieran controlados por un reloj. La señal de control que define estos intervalos de tiempo suele denominarse señal de reloj *clock*. El nombre *Clk* pretende reflejar la naturaleza de esta señal.

Los circuitos de este tipo, que usan una señal de control, se denominan *latches asíncronos*. Puesto que nuestro circuito cuenta con la capacidad de establecerse en 1 e inicializarse en 0, se le llama *latch SR asíncrono* [*S* por *set* y *R* por *reset*, establecerse e inicializarse, respectivamente]. En la figura 7.6*b* se describe su comportamiento. Ahí se define el estado de la salida Q en el tiempo $t + 1$, es decir, $Q(t + 1)$, como una función de las entradas *S*, *R* y *Clk*. Cuando $Clk = 0$, el latch permanece en el estado en que está en el instante *t*, es decir, $Q(t)$, independientemente de los valores de *S* y *R*. Esto se indica al especificar $S = x$ y $R = x$, donde x significa que el valor de la señal puede ser 0 o 1. (Recuérdese que ya empleamos esta notación en el capítulo 4.) Cuando $Clk = 1$, el circuito se comporta como el latch básico de la figura 7.5. Se establece en 1 mediante $S = 1$ y se inicializa por medio de $R = 1$. En la última fila de la tabla, donde $S = R = 1$, se muestra que el estado $Q(t + 1)$ no está definido porque no sabemos si será 0 o 1, lo cual corresponde a la situación descrita en la sección 7.1 junto con el diagrama de tiempo de la figura 7.5 en el tiempo t_{10}. En este tiempo las entradas *S* y *R* pasan de 1 a 0, lo que ocasiona el comportamiento oscilatorio que ya explicamos. Si $S = R = 1$, esta situación ocurrirá en cuanto *Clk* pase de 1 a 0. Para asegurar una operación significativa del latch SR asíncrono es esencial evitar la posibilidad de tener las dos entradas *S* y *R* iguales a 1 cuando *Clk* cambia de 1 a 0.

Un diagrama de tiempo para el latch SR asíncrono se muestra en la figura 7.6*c*. *Clk* se presenta como una señal periódica igual a 1 en intervalos de tiempo regulares para indicar que tal es casi siempre la forma en que aparece la señal de reloj en un sistema real. En el diagrama se advierte el efecto de varias combinaciones de valores de señal. Nótese que hemos etiquetado una salida como Q y la otra como su complemento $\overline{Q}$, en vez de Q_a y Q_b como en la figura 7.5. Puesto que el modo indefinido, donde $S = R = 1$, debe evitarse en la práctica, la operación normal del latch tendrá las salidas como complementos unas de las otras. Además, con frecuencia decimos que el latch se establece en 1 cuando $Q = 1$, y se *inicializa* cuando $Q = 0$. En la figura 7.6*d* se muestra el símbolo gráfico para el latch SR asíncrono.

7.2.1 LATCH SR ASÍNCRONO CON COMPUERTAS NAND

Hasta ahora hemos implementado el latch básico con compuertas NOR con acoplamiento cruzado. También podemos construir el latch con compuertas NAND. Si utilizamos este método es posible implementar el latch SR asíncrono como se muestra en la figura 7.7. El comportamiento de este circuito se describe en la tabla de la figura 7.6*b*. Obsérvese que en este circuito el reloj

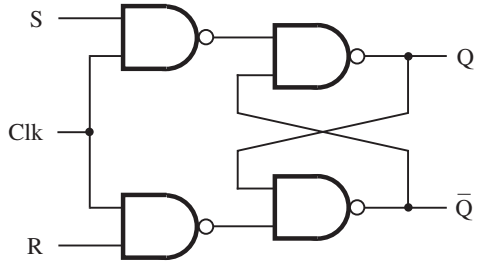


Figura 7.7 Latch SR asíncrono con compuertas NAND.

está controlado por compuertas NAND en vez de AND. Adviértase también que las entradas *S* y *R* están invertidas en comparación con el circuito de la figura 7.6*a*. El circuito con compuertas NAND requiere menos transistores que el circuito con compuertas AND, por lo cual optaremos por usar el circuito de la figura 7.7 en lugar del de la figura 7.6*a*.

7.3 LATCH D ASÍNCRONO

En la sección 7.2 presentamos el latch SR asíncrono y mostramos de qué manera usarlo como el elemento de memoria en el sistema de alarma de la figura 7.1. Este latch es útil para muchas otras aplicaciones. En esta sección describimos otro latch asíncrono aún más útil en la práctica. Tiene una sola entrada de datos, llamada *D*, y almacena el valor de esta entrada bajo el control de una señal de reloj. Se llama *latch D asíncrono*.

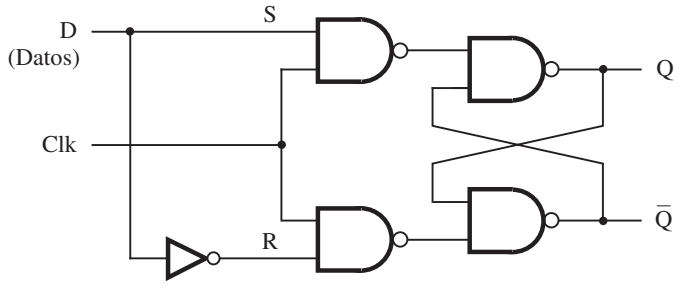
Para generar la necesidad de un latch D asíncrono, considérese la unidad de sumador/restador expuesta en el capítulo 5 (figura 5.13). Cuando describimos cómo se usa ese circuito para sumar números, no explicamos lo que es probable que ocurra con los bits de suma producidos por el sumador. Las unidades de sumador/restador se emplean como parte de una computadora. El resultado de una operación de suma o resta con frecuencia se utiliza como un operando en una operación posterior. Por tanto, es preciso recordar los valores de los bits de suma generados por el sumador en caso que se requieran de nuevo. Podríamos pensar en usar los latches básicos para recordar esos bits, un latch por bit. En este contexto, en vez de decir que un latch recuerda el valor de un bit, resulta más claro decir que el latch *almacena* el valor del bit, o simplemente "almacena el bit". Debemos pensar en el latch como un elemento de almacenamiento.

Pero ¿podemos obtener la operación buscada con latches básicos? Sin duda, es posible inicializar todos los latches antes que la operación de adición comience. Luego esperaríamos que al conectar un bit de suma a la entrada *S*, el latch se establezca en 1 si el bit de suma tiene el valor de 1; de lo contrario, el latch permanecería en el estado 0. Esto funcionaría bien si todos los bits de suma fueran 0 al comienzo de la operación de adición y, después de cierto retraso de propagación a través del sumador, algunos de esos bits se vuelven iguales a 1 para proporcionar la suma buscada. Lamentablemente, los retrasos de propagación que hay en el circuito del sumador causan un problema mayor en este arreglo. Supóngase que usamos un sumador con acarreo en cascada. Cuando las entradas *X* y *Y* se aplican al sumador, las salidas de la suma pueden alternar entre 0 y 1 varias veces a medida que los acarreos pasan por el circuito. Esta situación se ilustró en el diagrama de tiempo de la figura 5.21. El problema es que cuando conectamos un bit de suma a la entrada *S* de un latch, entonces si temporalmente el bit de suma es 1 y luego se establece en 0 en el resultado final, el latch permanecerá en 1 por error.

El problema ocasionado por los valores alternos de los bits de suma en el sumador podría resolverse si empleamos latches SR asíncronos en vez de latches básicos. Luego podríamos arreglar que la señal de reloj sea 0 durante el tiempo que requiera el sumador para producir una suma correcta. Después de permitir el retraso de propagación máximo en el circuito sumador, el reloj pasaría a 1 para almacenar los valores de los bits de suma en los latches asíncronos. En cuanto los valores se hayan almacenado, el reloj puede regresar a 0, lo cual asegura que los valores almacenados se conservarán hasta la siguiente vez que el reloj pase por 1. Para lograr la operación buscada también podríamos hacer que todos los latches se inicialicen en 0 antes de cargar los valores del bit de suma en ellos. Ésta es una forma poco práctica de encarar el problema y es preferible usar latches D asíncronos.

En la figura 7.8*a* se muestra el circuito para un latch D asíncrono. Se basa en el latch SR asíncrono, pero en vez de utilizar entradas S y R por separado, sólo tiene una entrada de datos, D. Por conveniencia hemos etiquetado los puntos del circuito equivalentes a las entradas S y R. Si $D = 1$, entonces $S = 1$ y $R = 0$, lo que obliga al latch a entrar en el estado Q = 1. Si $D = 0$, entonces $S = 0$ y $R = 1$, lo cual hace que Q = 0. Desde luego, los cambios de estado sólo ocurren cuando $Clk = 1$.

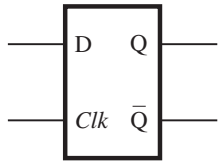
Es importante observar que en este circuito resulta imposible que se presente el problema de $S = R = 1$. En el latch D asíncrono, la salida Q simplemente sigue al valor de la entrada D cuando $Clk = 1$. En cuanto Clk pasa a 0, el estado del latch se congela hasta la siguiente vez que la señal de reloj pasa a 1.Por consiguiente, el latch D asíncrono almacena el valor de la entrada D



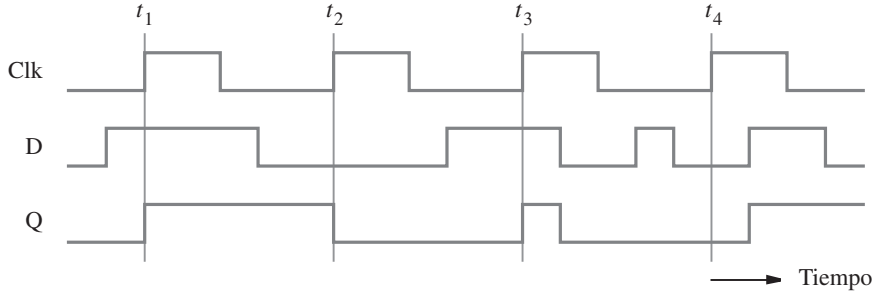
a) Circuito

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

b) Tabla característica



c) Símbolo gráfico



d) Diagrama de tiempo

Figura 7.8 Latch D asíncrono.

visto en el instante que el reloj cambia de 1 a 0. En la figura 7.8 además se proporciona la tabla característica, el símbolo gráfico y el diagrama de tiempo para el latch D asíncrono.

El diagrama de tiempo ilustra lo que ocurre si la señal *D* cambia mientras *Clk* = 1. Durante el tercer pulso del reloj, comenzando en t_3, la salida Q cambia a 1 porque *D* = 1. Pero a mitad del camino que recorre el pulso *D* pasa a 0, lo cual ocasiona que Q se establezca en 0. Este valor de Q se almacena cuando *Clk* cambia a 0. Ahora ya no ocurre ningún cambio en el estado del latch hasta el siguiente pulso del reloj en t_4. Lo más importante que debe advertirse es que mientras el reloj tiene el valor 1, la salida Q sigue a la entrada *D*. Pero cuando el reloj tiene el valor 0, la salida Q no puede cambiar. En el capítulo 3 vimos que los valores lógicos se implementan como niveles de voltaje altos y bajos. Puesto que la salida del latch D asíncrono está controlada por el nivel de la entrada del reloj, se dice que el latch es *sensible al nivel*. Los circuitos de las figuras 7.6 a 7.8 son sensibles al nivel. En la sección 7.4 mostraremos que es posible diseñar elementos de almacenamiento para los que la salida sólo cambia en el instante en que el reloj cambia de un valor a otro. Se dice que tales circuitos se *disparan por flanco*.

En este punto debemos considerar de nuevo el circuito de la figura 7.3. Al examinarlo detalladamente veremos que se comporta igual que el circuito de la figura 7.8*a*. Las entradas *Data* y *Load* corresponden a las entradas *D* y *Clk*, respectivamente. *Output*, que tiene el mismo valor de señal que el punto *A*, corresponde a la salida Q. El punto *B* corresponde a $\overline{Q}$. Por tanto, el circuito de la figura 7.3 también es un latch D asíncrono. Una ventaja de este circuito es que puede implementarse con menos transistores que el circuito de la figura 7.8*a*.

7.3.1 EFECTOS DE LOS RETRASOS DE PROPAGACIÓN

En el análisis anterior ignoramos los efectos de los retrasos de propagación. En los circuitos prácticos es esencial tomarlos en cuenta. Considérese el latch D asíncrono de la figura 7.8*a*. Almacena el valor de la entrada *D* que se presenta en el momento en que la señal de reloj cambia de 1 a 0. Opera en forma apropiada si la señal *D* es estable (es decir, si no cambia) en el instante en que *Clk* pasa de 1 a 0. Pero puede conducir a resultados impredecibles si la señal *D* también cambia en ese tiempo. Por ende, el diseñador de un circuito lógico que genera la señal *D* debe cerciorarse que ésta sea estable cuando ocurra el cambio crítico en la señal de reloj.

En la figura 7.9 se ilustra la región de temporización crítica. El tiempo mínimo que la señal *D* debe permanecer estable antes del flanco negativo de la señal *Clk* se llama *tiempo de preparación*, t_{su}, del latch. El tiempo mínimo que la señal *D* debe permanecer estable después del flanco

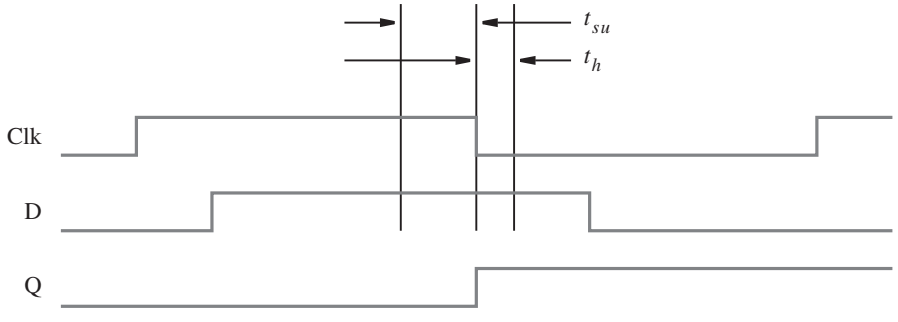


Figura 7.9 Tiempos de preparación y de espera.

negativo de la señal *Clk* recibe el nombre de *tiempo de espera*, t_h, del latch. Los valores de t_{su} y t_h dependen de la tecnología empleada. Los fabricantes de los chips de circuitos integrados proporcionan esta información en las hojas de datos que describen sus chips. Los valores típicos para la tecnología CMOS son $t_{su} = 3$ ns y $t_h = 2$ ns. En la sección 7.13 daremos ejemplos de la manera en que los tiempos de preparación y de espera afectan la velocidad de operación de los circuitos. En la sección 10.3.3 se explica el comportamiento de los elementos de almacenamiento cuando se incumplen los tiempos de preparación o de espera.

7.4 FLIP-FLOPS D MAESTRO-ESCLAVO Y DISPARADO POR FLANCO

En los latches sensibles al nivel, el estado del latch sigue cambiando de acuerdo con los valores de las señales de entrada mientras la señal de reloj está activa (igual a 1 en nuestros ejemplos). Como se verá en las secciones 7.8 y 7.9, también existe la necesidad de elementos de almacenamiento que puedan cambiar sus estados no más de una vez durante un ciclo del reloj. Estudiaremos dos tipos de circuitos que presentan este comportamiento.

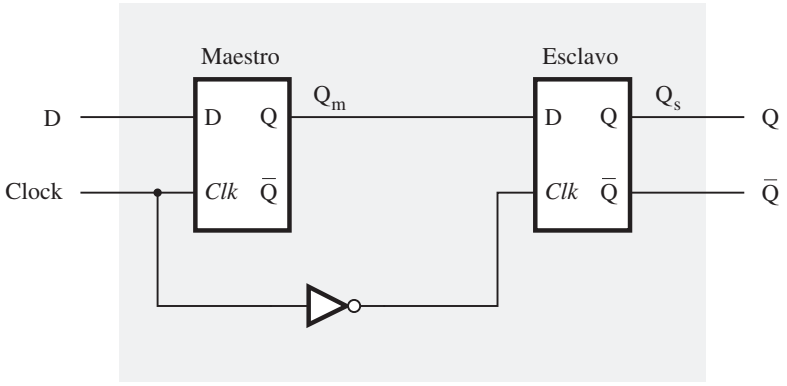
7.4.1 FLIP-FLOP D MAESTRO-ESCLAVO

Considérese el circuito de la figura 7.10*a*, el cual se compone de dos latches D asíncronos. El primero, llamado *maestro*, cambia su estado mientras *Clock* = 1. El segundo, denominado *esclavo*, lo hace mientras *Clock* = 0. El funcionamiento del circuito es tal que cuando el reloj está en nivel alto, el latch maestro sigue el valor de la señal de entrada *D* y el esclavo no cambia. Por tanto, el valor de Q_m sigue cualquier cambio en *D* y el de Q_s permanece constante. Cuando la señal de reloj cambia a 0, la etapa de maestro deja de seguir los cambios en la entrada *D*. Al mismo tiempo, la etapa de esclavo responde al valor de la señal Q_m y por consiguiente cambia de estado. Como Q_m no cambia mientras *Clock* = 0, la etapa de esclavo puede sufrir cuando mucho un cambio de estado durante un ciclo del reloj. Desde el punto de vista del observador externo, es decir, del circuito conectado a la salida de la etapa de esclavo, el circuito maestro-esclavo cambia su estado en el flanco del reloj que va a ser negativo. El *flanco negativo* es el flanco donde la señal de reloj cambia de 1 a 0. Independientemente del número de cambios en la entrada *D* a la etapa de maestro durante un ciclo del reloj, el observador de la señal Q_s verá sólo el cambio que corresponde a la entrada *D* en el flanco negativo del reloj.

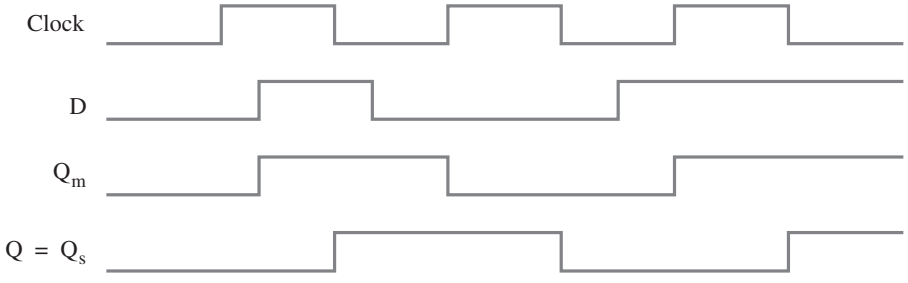
El circuito de la figura 7.10 se llama *flip-flop D maestro-esclavo*. El término *flip-flop* indica un elemento de almacenamiento que cambia el estado de su salida en el flanco de una señal controladora de reloj. El diagrama de tiempo de este flip-flop se muestra en la figura 7.10*b*. En la figura 7.10*c* aparece el símbolo gráfico correspondiente; en él usamos el signo $>$ para indicar que el flip-flop responde al "flanco activo" del reloj. Colocamos una burbuja en la entrada del reloj para indicar que el flanco activo de este circuito en particular es el flanco negativo.

7.4.2 FLIP-FLOP D DISPARADO POR FLANCO

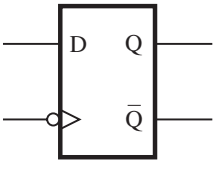
La salida del flip-flop D maestro-esclavo de la figura 7.10*a* responde al flanco negativo de la señal de reloj. El circuito puede modificarse para que responda al flanco positivo del reloj al conectar la etapa de esclavo directamente al reloj y la etapa de maestro al complemento de éste.



a) Circuito



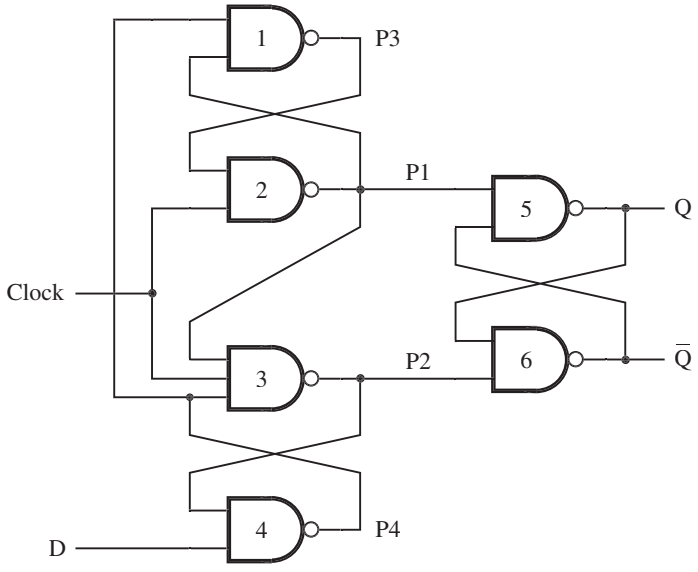
b) Diagrama de tiempo



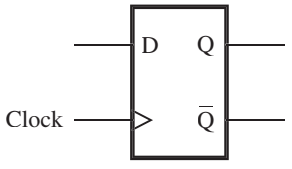
c) Símbolo gráfico

Figura 7.10 Flip-flop D maestro-esclavo.

En la figura 7.11*a* se presenta un circuito diferente que realiza la misma tarea. Requiere sólo seis compuertas NAND y, por consiguiente, menos transistores. El funcionamiento del circuito es como sigue. Cuando *Clock* = 0, las salidas de las compuertas 2 y 3 están en nivel alto. Por tanto, $P1 = P2 = 1$, lo que mantiene el latch de salida, compuesto por las compuertas 5 y 6, en su estado actual. Al mismo tiempo, la señal $P3$ es igual a D, y $P4$ es igual a su complemento $\overline{D}$.



a) Circuito



b) Símbolo gráfico

Figura 7.11 Un flip-flop D disparado por el flanco positivo.

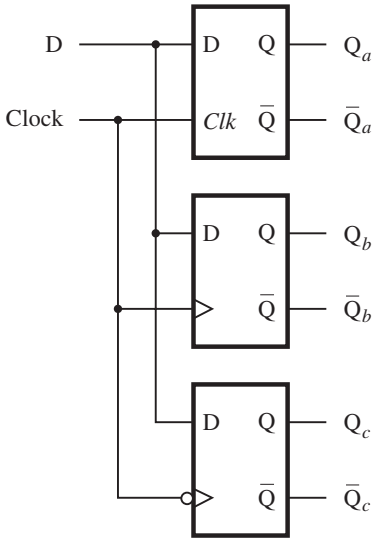
Cuando *Clock* cambia a 1, ocurren los cambios siguientes. Los valores de $P3$ y $P4$ se transmiten a través de las compuertas 2 y 3 para hacer que $P1 = \overline{D}$ y $P2 = D$, lo que establece $Q = D$ y $\overline{Q} = \overline{D}$. Para que funcionen de manera confiable, $P3$ y $P4$ deben hallarse estables cuando el reloj cambie de 0 a 1. Por tanto, el tiempo de preparación del flip-flop es igual al retraso de la entrada D a través de las compuertas 4 y 1 hacia $P3$. El tiempo de espera está dado por el retraso de la compuerta 3 porque una vez que $P2$ está estable, los cambios en D ya no importan.

Para que haya un funcionamiento adecuado es preciso mostrar que, después que *Clock* cambie a 1, cualquier cambio posterior en D no afectará el latch de salida siempre que $Clock = 1$. Hay que considerar dos casos. Supóngase primero que $D = 0$ en el flanco positivo del reloj. Entonces $P2 = 0$, lo cual mantendrá la salida de la compuerta 4 igual a 1 siempre que $Clock = 1$, independientemente del valor de la entrada D. El segundo caso es cuando $D = 1$ en el flanco positivo del reloj. Entonces $P1 = 0$, lo que obliga a que las salidas de las compuertas 1 y 3 sean iguales a 1, independientemente de la entrada D. Por ende, el flip-flop ignora los cambios en la entrada D mientras $Clock = 1$.

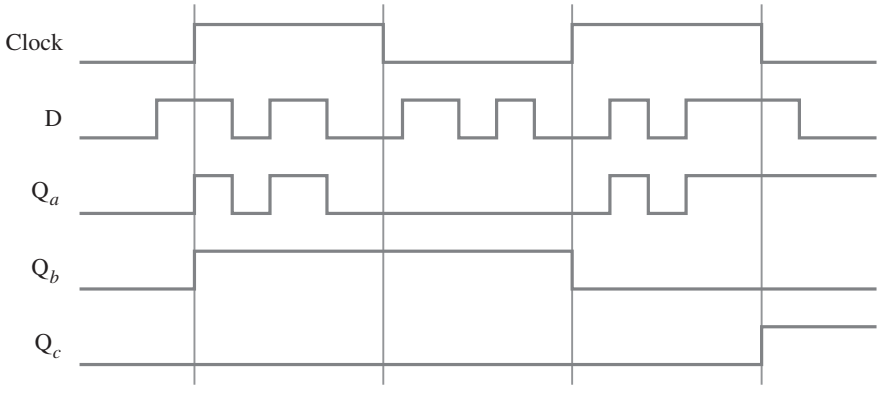
En la figura 7.11*b* se presenta un símbolo gráfico para este flip-flop. La entrada del reloj indica que el flanco positivo del reloj es el activo. Un circuito similar, construido con compuertas NOR, puede usarse como un flip-flop disparado por el flanco negativo.

Elementos de almacenamiento sensibles al nivel *versus* elementos disparados por flanco

En la figura 7.12 se muestran tres tipos de elementos de almacenamiento manejados por los mismos datos y entradas del reloj. El primer elemento es un latch D asíncrono, sensible al nivel. El segundo es un flip-flop D disparado por el flanco positivo y el tercero es un flip-flop D disparado por el flanco negativo. Para acentuar las diferencias entre estos elementos de almacenamiento,



a) Circuito



b) Diagrama de tiempo

Figura 7.12 Comparación de elementos de almacenamiento D sensibles al nivel con elementos de almacenamiento D disparados por flanco.

la entrada *D* cambia sus valores más de una vez durante cada semiciclo del reloj. Obsérvese que el latch D asíncrono sigue a la entrada *D* mientras el reloj esté en nivel alto. El flip-flop disparado por el flanco positivo responde sólo al valor de *D* cuando el reloj cambia de 0 a 1. El flip-flop disparado por el flanco negativo responde sólo al valor de *D* cuando el reloj cambia de 1 a 0.

7.4.3 FLIP-FLOPS D CON CLEAR Y PRESET

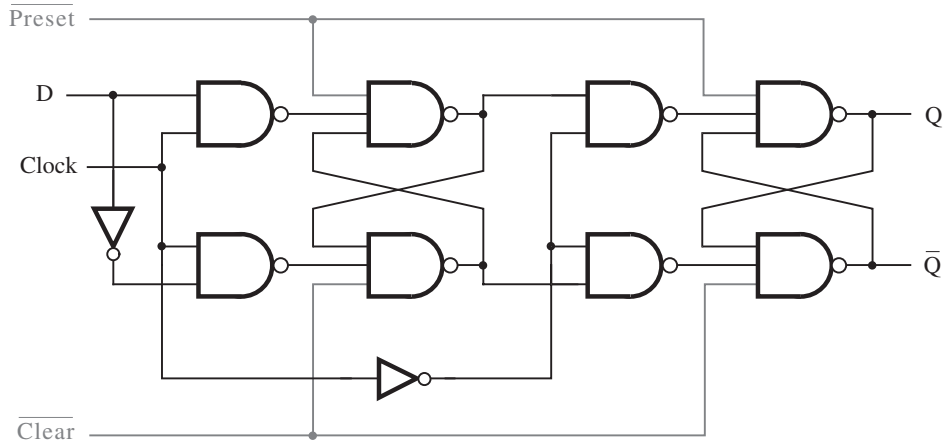
Los flip-flops suelen utilizarse para implementar circuitos que pueden tener muchos estados posibles, donde la respuesta del circuito depende no sólo de los valores que hay en las entradas sino también de los valores del estado particular en que se halla el circuito en ese instante. En el capítulo siguiente analizaremos una forma general de tales circuitos. Un ejemplo simple es un circuito contador que cuenta el número de ocurrencias de algún evento, tal vez el paso del tiempo. Estudiaremos los contadores con detalle en la sección 7.9. Un contador comprende una serie de flip-flops, cuyas salidas se interpretan como un número. El circuito contador debe tener la capacidad de aumentar o disminuir este número. También es importante poder forzarlo a entrar en un estado inicial conocido (*count*). Lógicamente, debe ser posible borrar el contador y dejarlo en cero, lo que significa que todos los flip-flops deben tener Q = 0. También es útil poder preestablecer cada flip-flop en Q = 1 para insertar una cuenta específica como el valor inicial del contador. Estas características pueden incorporarse a los circuitos de las figuras 7.10 y 7.11 como sigue.

En la figura 7.13*a* se muestra una implementación del circuito de la figura 7.10a usando compuertas NAND. La etapa de maestro es simplemente el latch D asíncrono de la figura 7.8*a*. En vez de usar otro latch del mismo tipo para la etapa de esclavo, podemos usar el latch SR asíncrono ligeramente más simple de la figura 7.7. Con ello se elimina una compuerta NOT del circuito.

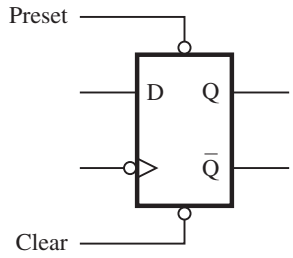
Una forma sencilla de proporcionar la capacidad de borrar (*clear*) y preestablecer (*preset*) consiste en añadir una entrada adicional a cada compuerta NAND en los latches con acoplamiento cruzado, como se indica en gris oscuro. Colocar un 0 en la entrada *Clear* hará que el flip-flop entre en el estado Q = 0. Si *Clear* = 1, entonces esta entrada no producirá efecto alguno en las compuertas NAND. De forma semejante, *Preset* = 0 obliga a que el flip-flop entre en el estado Q = 1, mientras que *Preset* = 1 no produce efecto. Para indicar que las entradas *Clear* y *Preset* están activas cuando sus valores son 0 ponemos una línea arriba de los nombres en la figura. Cabe señalar que el circuito que utiliza este flip-flop no debe intentar forzar que tanto *Clear* como *Preset* se establezcan en 0 al mismo tiempo. Un símbolo gráfico para este flip-flop se muestra en la figura 7.13*b*.

Una modificación similar puede hacerse en el flip-flop disparado por flanco de la figura 7.11*a*, como se indica en la figura 7.14*a*. De nuevo, tanto las entradas *Clear* como *Preset* están activas en nivel bajo. No alteran el flip-flop cuando son iguales a 1.

En los circuitos de las figuras 7.13*a* y 7.14*a*, el efecto de una señal baja en la entrada *Clear* o en *Preset* es inmediato. Por ejemplo, si *Clear* = 0 entonces el flip-flop entra en el estado Q = 0 inmediatamente, con independencia del valor de la señal de reloj. En un circuito como éste, donde la señal *Clear* sirve para borrar un flip-flop sin importar la señal de reloj, decimos que el flip-flop tiene un *borrado asíncrono*. En la práctica, a menudo es preferible borrar los flip-flops en el flanco activo del reloj. Este *borrado síncrono* puede lograrse como se muestra en la figura 7.15. El flip-flop opera con normalidad cuando la entrada *Clear* es igual a 1; pero si pasa a 0, entonces en el siguiente flanco negativo del reloj el flip-flop se establecerá en 0. Examinaremos el borrado de los flips-flops con más detalle en la sección 7.10.



a) Circuito

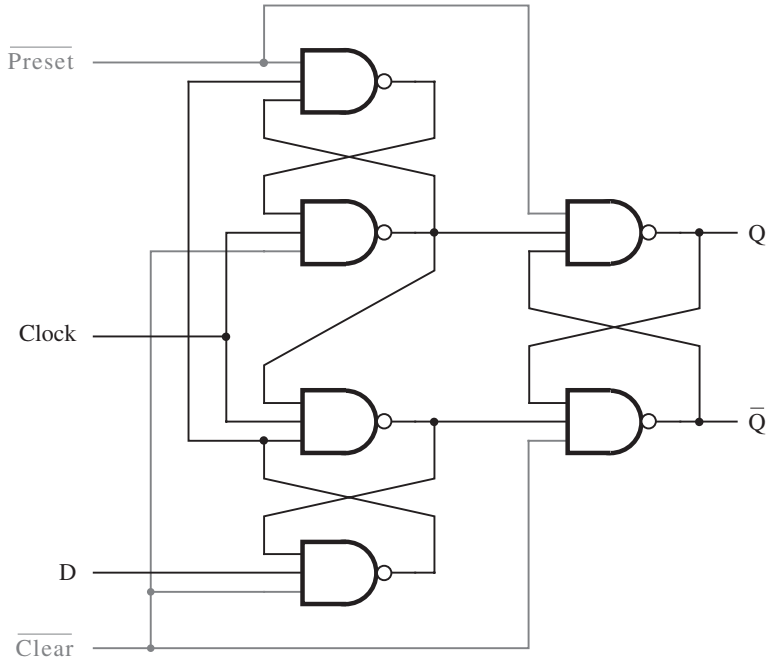


b) Símbolo gráfico

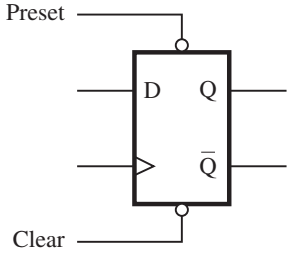
Figura 7.13 Flip-flop D maestro-esclavo con Clear y Preset.

7.5 FLIP-FLOP T

El flip-flop D es un elemento de almacenamiento polifacético que sirve para muchos propósitos. Al incluir un sistema de circuitos lógico simple para manejar su entrada, el flip-flop D puede parecer un elemento de almacenamiento distinto. Una modificación interesante se presenta en la figura 7.16*a*. Este circuito usa un flip-flop D disparado por el flanco positivo. Las conexiones de *retroalimentación* hacen que la señal de entrada *D* sea igual a cualquier valor de Q o $\overline{Q}$ bajo el control de la señal etiquetada con *T*. En cada flanco positivo del reloj, el flip-flop puede cambiar su estado Q(*t*). Si $T = 0$, entonces $D = Q$ y el estado seguirá siendo el mismo, es decir, $Q(t + 1) = Q(t)$. Pero si $T = 1$, entonces $D = \overline{Q}$ y el nuevo estado será $Q(t + 1) = \overline{Q}(t)$. Por tanto, la operación general del circuito es que éste conserva su estado presente si $T = 0$ y lo invierte si $T = 1$.



a) Circuito



b) Símbolo gráfico

Figura 7.14 Flip-flop D disparado por el flanco positivo con Clear y Preset.

El funcionamiento del circuito se especifica como una tabla característica en la figura 7.16*b*. Cualquier circuito que implemente esta tabla se llama *flip-flop T*. El nombre flip-flop T proviene del comportamiento del circuito, el cual "alterna entre dos estados" (*toggle*) cuando $T = 1$. La función toggle hace que el flip-flop T sea útil para construir circuitos contadores, como veremos en la sección 7.9.

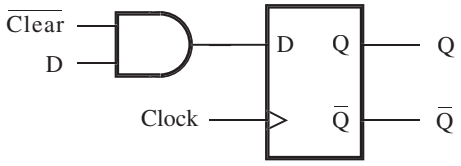
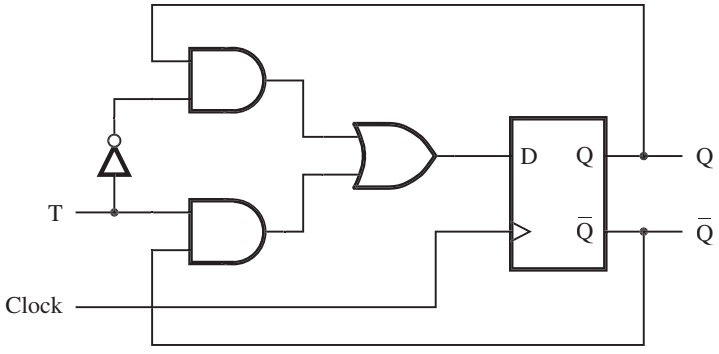


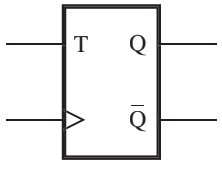
Figura 7.15 Inicialización síncrona para un flip-flop D.



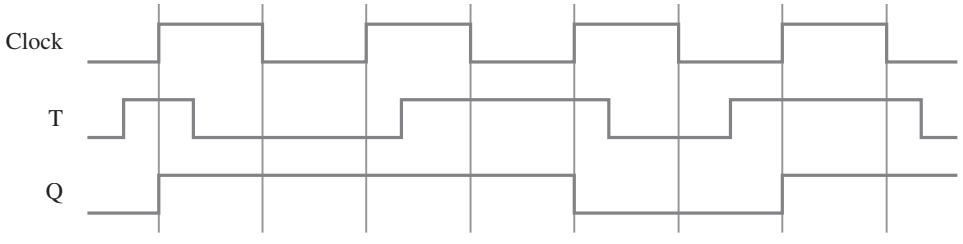
a) Circuito

T	Q(*t* + 1)
0	Q(*t*)
1	$\overline{Q}(t)$

b) Tabla característica



c) Símbolo gráfico



d) Diagrama de tiempo

Figura 7.16 Flip-flop T.

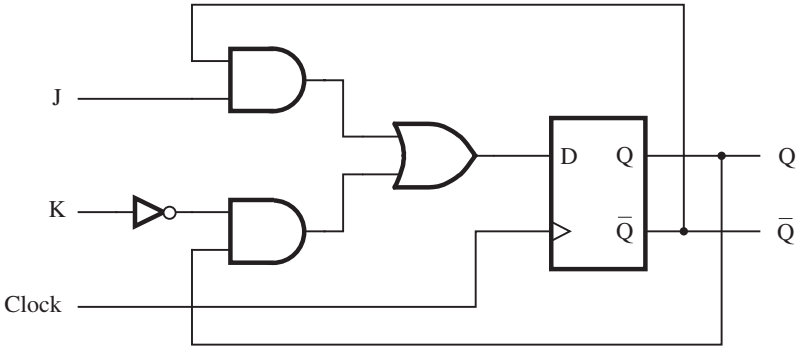
7.5.1 FLIP-FLOPS CONFIGURABLES

Para algunos circuitos un tipo de flip-flop puede llevar a una implementación más eficaz que otro tipo. En los chips de uso general como los PLD, los flip-flops que se proporcionan a veces son *configurables*, lo cual significa que un circuito de flip-flop puede configurarse para ser D, T o de otro tipo. Por ejemplo, en algunos PLD los flip-flops pueden configurarse ya sea como tipo D o T (véanse los problemas 7.6 y 7.8).

7.6 FLIP-FLOP JK

Otro circuito interesante puede derivarse de la figura 7.16*a*. En vez de usar una sola entrada de control, *T*, podemos usar dos entradas, *J* y *K*, como se indica en la figura 7.17*a*. Para este circuito la entrada *D* se define como

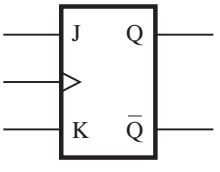
$$D = J\overline{Q} + \overline{K}Q$$



a) Circuito

J	K	Q(*t* + 1)
0	0	Q(*t*)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

b) Tabla característica



c) Símbolo gráfico

Figura 7.17 Flip-flop JK.

En la figura 7.17*b* se presenta la tabla característica correspondiente. El circuito se llama *flip-flop JK*. Combina el comportamiento de los flip-flops SR y T de una manera útil. Se comporta como el flip-flop SR, donde $J = S$ y $K = R$, para todos los valores de entrada excepto $J = K = 1$. En este último caso, el cual debe evitarse en el flip-flop SR, el flip-flop JK alterna su estado del mismo modo que el flip-flop T.

El flip-flop JK es un circuito multifuncional. Puede usarse para almacenamiento sencillo, como los flip-flops D y SR. Pero también sirve como un flip-flop T si se conectan juntas las entradas *J* y *K*.

7.7 RESUMEN DE TERMINOLOGÍA

Hemos usado la terminología más común. Pero el lector debe estar consciente de que puede haber diferentes interpretaciones de los términos *latch* y *flip-flop* en otras obras. Nuestra terminología puede resumirse como sigue:

El **latch básico** es una conexión de retroalimentación de dos compuertas NOR o NAND, las cuales pueden almacenar un bit de información. Este latch puede establecerse en 1 utilizando la entrada *S* e inicializarse en 0 con la entrada *R*.

El **latch asíncrono** es un latch básico que incluye compuertas de entrada y una señal de entrada de control. El latch conserva su estado actual cuando la entrada de control es igual a 0. Su estado puede cambiar cuando la señal de control es igual a 1. En nuestra exposición nos referimos a la entrada de control como el reloj. Consideramos dos tipos de latches asíncronos:

- El **latch SR asíncrono** usa las entradas *S* y *R* para establecer el latch en 1 o inicializarlo en 0, respectivamente.
- El **latch D asíncrono** usa la entrada *D* para obligar al latch a entrar en un estado que tiene el mismo valor lógico que la entrada *D*.

Un flip-flop es un elemento de almacenamiento basado en el principio del latch asíncrono, el cual únicamente puede cambiar su estado de salida en el flanco de la señal de reloj controladora. Consideramos dos tipos de flip-flop:

- El **flip-flop disparado por flanco**, que se ve afectado sólo por los valores de entrada presentes cuando el reloj está en el flanco activo.
- El **flip-flop maestro-esclavo** se construye con dos latches asíncronos. La etapa de maestro está activa durante la mitad del ciclo del reloj, y la etapa de esclavo durante la otra mitad. El valor de salida del flip-flop cambia en el flanco del reloj que activa la transferencia en la etapa de esclavo. El flip-flop maestro-esclavo es disparado por flanco o sensible al nivel. Si la etapa de maestro es un latch D asíncrono, entonces se comporta como flip-flop disparado por flanco. Si la etapa de maestro es un latch SR asíncrono, entonces el flip-flop es sensible al nivel (véase el problema 7.19).

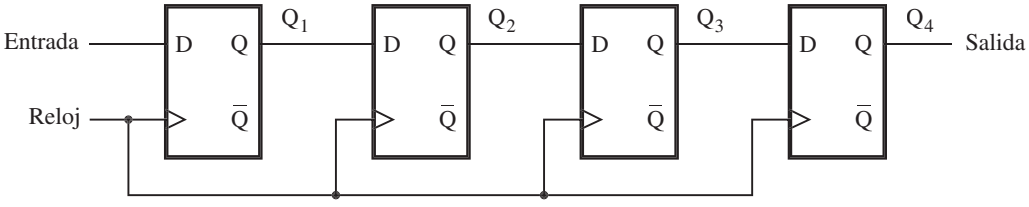
7.8 REGISTROS

Un flip-flop almacena un bit de información. Cuando un conjunto de *n* flip-flops se usa para almacenar *n* bits de información, como un número de *n* bits, nos referimos a él como un *registro*. Para cada flip-flop de un registro se usa un reloj común, y cada flip-flop funciona como se describió en la sección anterior. El término *registro* es simplemente una convención para referirse a las estructuras de *n* bits que se componen de flip-flops.

7.8.1 REGISTRO DE CORRIMIENTO

En la sección 5.6 explicamos que un número se multiplica por 2 si sus bits se desplazan una posición de un bit a la izquierda y se inserta un 0 como el nuevo bit menos significativo. De modo similar, el número se divide entre 2 si los bits se desplazan una posición de un bit a la derecha. Un registro que proporciona la capacidad para recorrer su contenido se llama *registro de corrimiento*.

En la figura 7.18*a* se muestra un registro de corrimiento de cuatro bits que se utiliza para desplazar su contenido una posición de un bit a la derecha. Los bits de datos se cargan en el registro de corrimiento en forma serial usando la entrada *In*. El contenido de cada flip-flop se transfiere al flip-flop siguiente



a) Circuito

	Entrada	Q_1	Q_2	Q_3	Q_4 = Salida
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

b) Una secuencia de muestra

Figura 7.18 Un registro de corrimiento simple.

en cada flanco positivo del reloj. En la figura 7.18*b* se presenta una ilustración de la transferencia; también se muestra lo que ocurre cuando los valores de la señal en *In* durante ocho ciclos del reloj consecutivos son 1, 0, 1, 1, 1, 0, 0 y 0, suponiendo que el estado inicial de todos los flip-flops es 0.

Para implementar un registro de corrimiento es necesario usar flip-flops disparados por flanco o maestro-esclavo. Los latches asíncronos sensibles al nivel no son adecuados, ya que un cambio en el valor de *In* se propagaría a través de más de un latch durante el tiempo que el reloj sea igual a 1.

7.8.2 REGISTRO DE CORRIMIENTO CON ACCESO EN PARALELO

En los sistemas de cómputo con frecuencia se requiere transferir elementos de datos de *n* bits. Esta transferencia puede realizarse transmitiendo todos los bits de una sola vez con *n* cables independientes, caso en el que decimos que la transferencia se realiza en *paralelo*. Pero también es posible transferir todos los bits por un solo cable, realizando la transferencia un bit a la vez, en *n* ciclos del reloj consecutivos. Nos referimos a este esquema como transferencia *serial*. Para transferir un elemento de datos de *n* bits en forma serial es posible usar un registro de corrimiento que puede cargarse con todos los *n* bits en paralelo (en un ciclo del reloj). Luego, durante los ciclos del reloj siguientes el contenido del registro puede desplazarse hacia afuera para la transferencia serial. También se requiere la operación inversa. Si los bits se reciben en forma serial, entonces después de *n* ciclos del reloj puede accederse al contenido del registro en paralelo como un elemento de *n* bits.

En la figura 7.19 se presenta un registro de corrimiento de cuatro bits que permite el acceso en paralelo. En vez de usar la conexión de registro de corrimiento normal, la entrada *D* de cada flip-flop se conecta a dos fuentes; una de ellas es el flip-flop precedente, el cual se necesita para que el registro de corrimiento opere. La otra fuente es la entrada externa que corresponde al bit que se va a cargar en el flip-flop como parte de la operación de carga en paralelo. La señal de control $\overline{Shift}/Load$ se usa para seleccionar el modo de operación. Si $\overline{Shift}/Load = 0$, entonces el circuito opera como un registro de corrimiento. Si $\overline{Shift}/Load = 1$, los datos de entrada en paralelo se cargan en el registro. En ambos casos la acción ocurre en el flanco positivo del reloj.

En la figura 7.19 elegimos etiquetar las salidas de los flip-flops como Q_3,..., Q_0 porque los registros de corrimiento con frecuencia se usan para alojar números binarios. Puede accederse en paralelo al contenido del registro al observar las salidas de todos los flip-flops. Los flip-flops también pueden accederse en forma serial, si se observan los valores de Q_0 durante ciclos del reloj consecutivos mientras el contenido se está desplazando. Un circuito en el que los datos pueden cargarse en serie y luego accederse en paralelo se llama convertidor de serial a paralelo. De forma similar, el tipo opuesto de circuito es un convertidor de paralelo a serial. El circuito de la figura 7.19 puede realizar estas dos funciones.

7.9 CONTADORES

En el capítulo 5 abordamos el tema de los circuitos que realizan operaciones aritméticas. Mostramos cómo pueden diseñarse los circuitos sumadores/restadores usando una estructura en cascada (*ripple-carry*) simple, que no es costosa pero sí lenta, o bien utilizando una estructura con acarreo hacia adelante más compleja, que es más cara aunque más rápida. En esta sección examinamos tipos especiales de operaciones de suma y resta usadas con el propósito de contar. En particular, queremos diseñar circuitos que puedan aumentar o disminuir un conteo en 1. Los circuitos

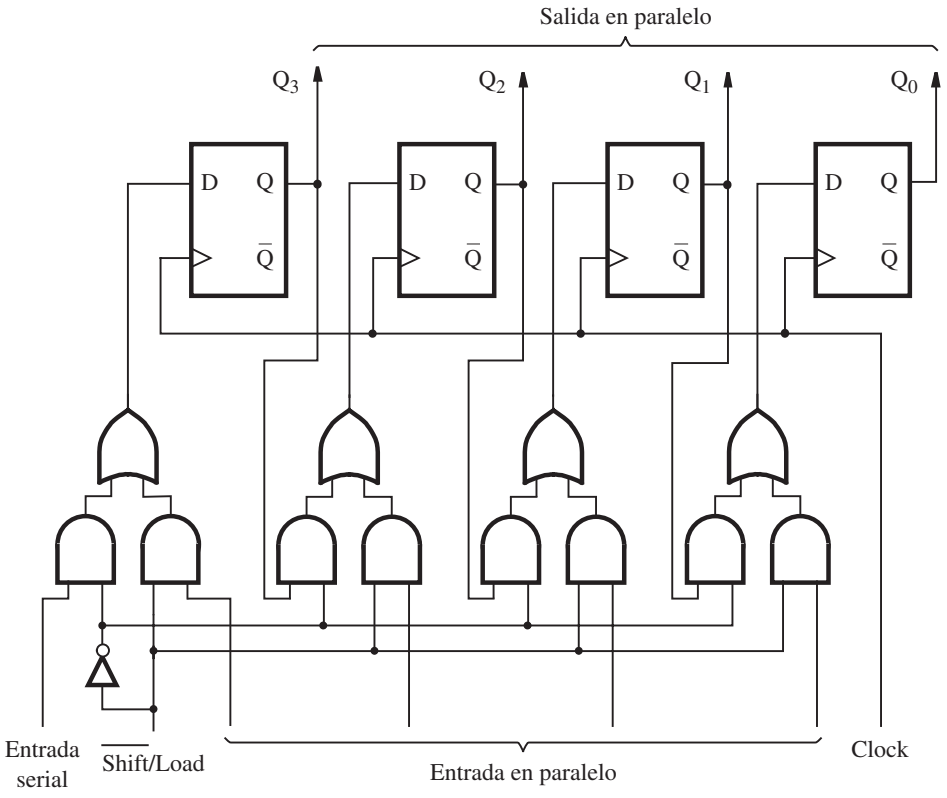


Figura 7.19 Registro de corrimiento con acceso en paralelo.

contadores se utilizan en los sistemas digitales para muchos fines. Pueden contar el número de ocurrencias de ciertos eventos, generar los intervalos de tiempo para el control de varias tareas en un sistema, llevar un registro del tiempo transcurrido entre eventos específicos, etcétera.

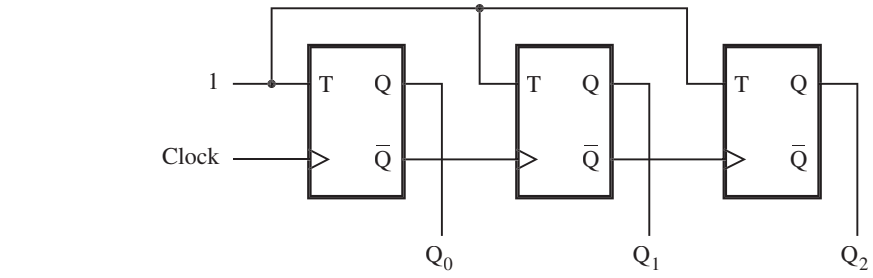
Los contadores pueden implementarse usando los circuitos sumadores/restadores estudiados en el capítulo 5 y los registros expuestos en la sección 7.8. Sin embargo, puesto que sólo necesitamos cambiar el contenido de un contador por 1, no es necesario usar circuitos muy elaborados. En vez de ello, empleamos circuitos mucho más sencillos que tienen un costo considerablemente menor. Mostraremos cómo se diseñan los circuitos contadores usando flip-flops T y D.

7.9.1 CONTADORES ASÍNCRONOS

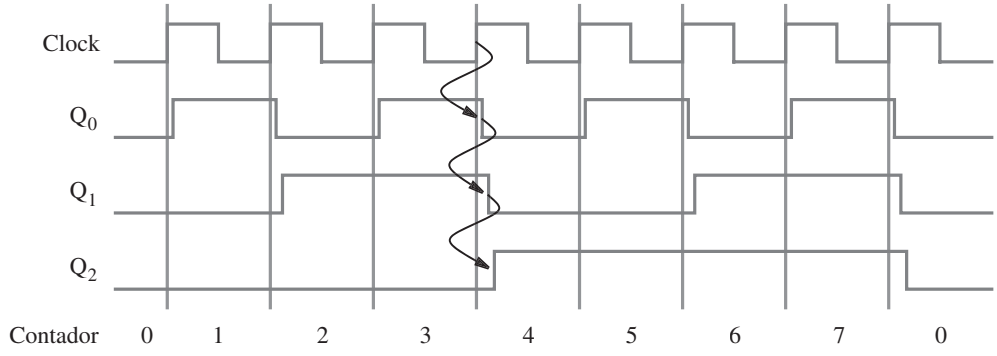
Los circuitos contadores más simples pueden construirse usando flip-flops T debido a que la función *toggle* se adecua de manera natural a la implementación de un conteo.

Contador ascendente con flip-flops T

En la figura 7.20*a* se describe un contador de tres bits que puede contar de 0 a 7. Las entradas del reloj de los tres flip-flops están conectadas en cascada. La entrada *T* de cada flip-flop está conectada a una constante 1, lo que significa que el estado del flip-flop se invertirá (cambiará a



a) Circuito



b) Diagrama de tiempo

Figura 7.20 Un contador ascendente de tres bits.

un segundo estado) en cada flanco positivo de su reloj. Estamos suponiendo que el propósito de este circuito es contar el número de pulsos que ocurren en la entrada principal llamada *Clock*. Por tanto, la entrada del reloj del primer flip-flop está conectada a la línea *Clock*. Las entradas del reloj de los otros dos flip-flops están manejadas por la salida $\overline{Q}$ del flip-flop anterior. Por consiguiente, alternan su estado siempre que el flip-flop precedente cambia su estado de $Q = 1$ a $Q = 0$, lo que resulta en el flanco positivo de la señal $\overline{Q}$.

En la figura 7.20*b* se muestra un diagrama de tiempo para el contador. El valor de Q_0 alterna una vez cada ciclo del reloj. El cambio ocurre poco tiempo después del flanco positivo de la señal *Clock*. El retraso es causado por la propagación a través del flip-flop. Como el segundo flip-flop está sincronizado por $\overline{Q}_0$, el valor de Q_1 cambia poco tiempo después del flanco negativo de la señal Q_0. De modo similar, el valor de Q_2 cambia poco tiempo después del flanco negativo de la señal Q_1. Si observamos los valores $Q_2Q_1Q_0$ como el conteo, entonces el diagrama de tiempo indica que la secuencia de conteo es 0, 1, 2, 3, 4, 5, 6, 7, 0, 1 y así sucesivamente. Este circuito es un contador módulo 8. Como cuenta en dirección ascendente, podemos llamarlo *contador ascendente*.

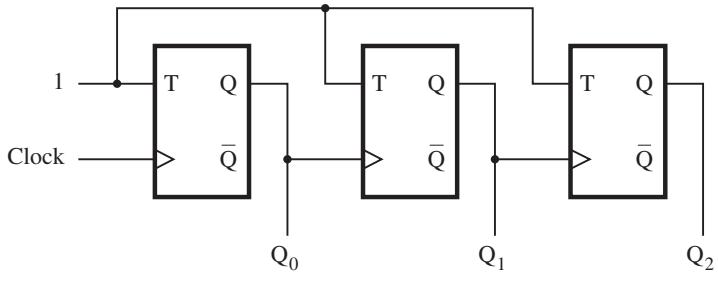
El contador de la figura 7.20*a* tiene tres *etapas*, y cada una consta de un solo flip-flop. Sólo la primera etapa responde directamente a la señal *Clock*; decimos que esta etapa está *sincronizada* con el reloj. Las otras dos etapas responden después de un retraso adicional. Por ejemplo, cuando *Count* = 3, el siguiente pulso del reloj hará que *Count* vaya a 4. Como se indica con las flechas

en el diagrama de tiempo de la figura 7.20*b*, este cambio requiere la alternación de los estados de los tres flip-flops. El cambio en Q_0 se observa sólo después de un retraso de propagación del flanco positivo de *Clock*. Los flip-flops Q_1 y Q_2 aún no han cambiado; por consiguiente, el conteo es $Q_2Q_1Q_0 = 010$ por un breve lapso. El cambio en Q_1 aparece después de un segundo retraso de propagación, momento en el que el conteo es 000. Finalmente, el cambio en Q_2 ocurre después de un tercer retraso, instante en el que el circuito llega a su estado estable y el conteo es 100. Este comportamiento se parece a la cascada de los acarreos en el circuito sumador de acarreo en cascada de la figura 5.6. El circuito de la figura 7.20*a* es un *contador asíncrono*, o un *contador en cascada*.

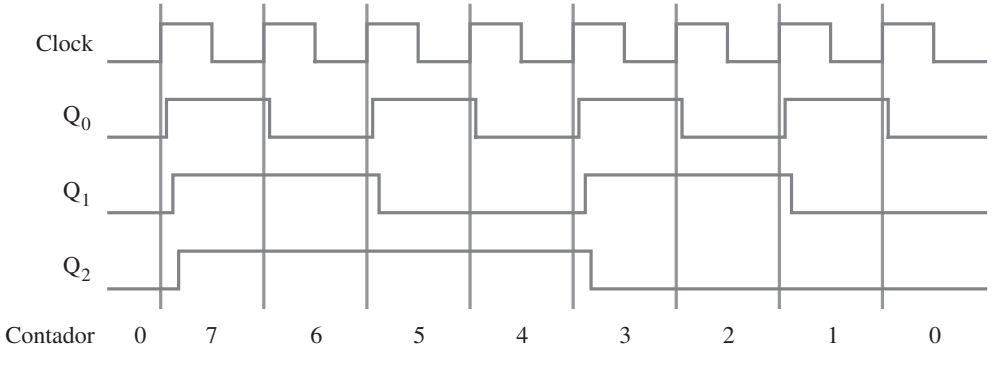
Contador descendente con flip-flops T

Una ligera modificación del circuito de la figura 7.20*a* se presenta en la figura 7.21*a*. La única diferencia es que en esta última figura las entradas del reloj del segundo y tercer flip-flops están manejadas por las salidas Q de las etapas precedentes, en vez de estarlo por las salidas $\overline{Q}$. En el diagrama de tiempo, dado en la figura 7.21*b*, se muestra que este circuito cuenta en la secuencia 0, 7, 6, 5, 4, 3, 2, 1, 0, 7 y así sucesivamente. Como cuenta en dirección descendente, decimos que es un *contador descendente*.

Es posible combinar la funcionalidad de los circuitos de las figuras 7.20*a* y 7.21*a* para formar un contador que pueda contar ascendente o descendentemente. Un contador como éste se



a) Circuito



b) Diagrama de tiempo

Figura 7.21 Un contador descendente de tres bits.

llama *contador ascendente/descendente*. Dejamos la deducción de este contador como ejercicio para el lector (problema 7.16).

7.9.2 CONTADORES SÍNCRONOS

Los contadores asíncronos de las figuras 7.20*a* y 7.21*a* son simples, pero no muy rápidos. Si se construye así un contador con un número grande de bits, las demoras causadas por el esquema de sincronización en cascada pueden volverse demasiado grandes para satisfacer los requisitos de desempeño deseados. Podemos construir un contador más rápido si sincronizamos todos los flip-flops al mismo tiempo aplicando el método descrito enseguida.

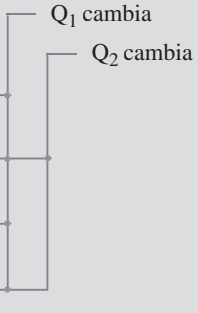
Contador síncrono con flip-flops T

En la tabla 7.1 se muestra el contenido de un contador ascendente de tres bits para ocho ciclos del reloj consecutivos, suponiendo que el conteo empieza en 0. Al observar el patrón de bits de cada fila de la tabla es claro que el bit Q_0 cambia en cada ciclo del reloj. El bit Q_1 cambia sólo cuando $Q_0 = 1$. El bit Q_2 cambia únicamente cuando Q_1 y Q_0 son iguales a 1. En general, para un contador ascendente de *n* bits, un flip-flop dado cambia su estado sólo cuando todos los flip-flops anteriores están en el estado Q = 1. Por consiguiente, si usamos flip-flops T para hacer el contador, entonces las entradas *T* se definen como

$$
\begin{aligned}
T_0 &= 1 \\
T_1 &= Q_0 \\
T_2 &= Q_0 Q_1 \\
T_3 &= Q_0 Q_1 Q_2 \\
&\vdots \\
T_n &= Q_0 Q_1 \cdots Q_{n-1}
\end{aligned}
$$

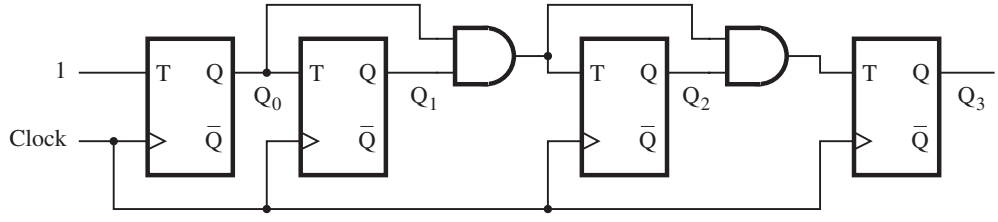
Tabla 7.1 Derivación del contador síncrono ascendente

Ciclo del reloj	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

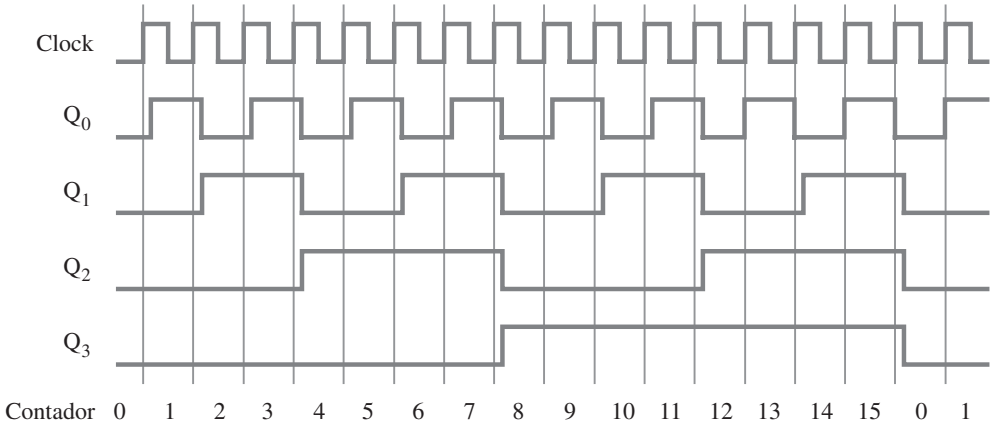


Un ejemplo de un contador de cuatro bits basado en estas expresiones aparece en la figura 7.22*a*. En vez de usar compuertas AND de tamaño aumentado para cada etapa, lo que puede suscitar problemas del factor de carga de entrada (*fan-in*), empleamos un arreglo factorizado, como se muestra en la figura. Este arreglo no reduce la respuesta del contador, ya que todos los flip-flops cambian sus estados después de un retraso de propagación desde el flanco positivo del reloj. Nótese que un cambio en el valor de Q_0 debe propagarse por varias compuertas AND para llegar a los flip-flops en las etapas superiores del contador, lo cual requiere cierta cantidad de tiempo. Este tiempo no debe exceder el periodo del reloj. En realidad, debe ser menor que tal periodo menos el tiempo de preparación para los flip-flops.

En la figura 7.22*b* se presenta un diagrama de tiempo. Se muestra que el circuito se comporta como un contador ascendente módulo 16. Debido a que todos los cambios ocurren con el mismo retraso después del flanco activo de la señal *Clock*, el circuito se llama *contador síncrono*.



a) Circuito



b) Diagrama de tiempo

Figura 7.22 Un contador síncrono ascendente de cuatro bits.

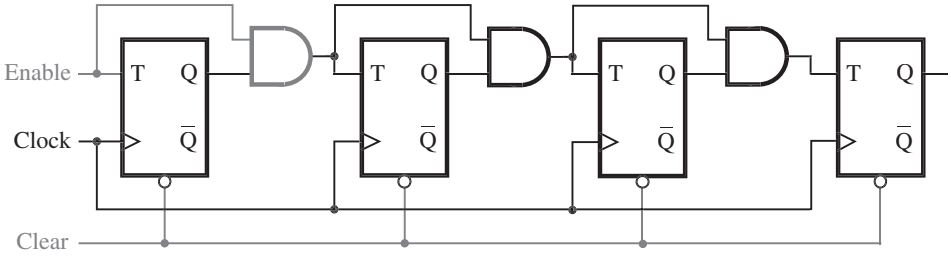


Figura 7.23 Inclusión de las capacidades Enable y Clear.

Capacidades Enable y Clear

Los contadores de las figuras 7.20 a 7.22 cambian su contenido en respuesta a cada pulso del reloj. Con frecuencia es aconsejable inhibir el conteo, de modo que siga en su etapa presente. Ello puede lograrse incluyendo una señal de control *Enable*, como se indica en la figura 7.23. El circuito es el contador de la figura 7.22, donde la señal *Enable* controla directamente la entrada *T* del primer flip-flop. Conectar además *Enable* a la cadena de la compuerta AND significa que si *Enable* = 0, entonces todas las entradas *T* serán iguales a 0. Si *Enable* = 1, entonces el contador funcionará como se explicó antes.

En muchas aplicaciones es preciso empezar con el conteo igual a cero, lo cual se logra fácilmente si los flip-flops pueden borrarse, como se explicó en la sección 7.4.3. Las entradas clear de todos los flip-flops pueden unirse y manejarse por medio de una entrada de control *Clear*.

Contador síncrono con flip-flops D

Si bien la función toggle hace que los flip-flops T sean una opción natural para la implementación de los contadores, también es posible construir contadores usando otros tipos de flip-flops. Los flip-flops JK pueden emplearse exactamente igual que los flip-flops T, ya que si las entradas *J* y *K* se unen, un flip-flop JK se convierte en un flip-flop T. Ahora consideraremos ocupar flip-flops D para este propósito.

No es claro cómo pueden utilizarse los flip-flops D para implementar un contador. Presentaremos un método formal para derivar estos circuitos en el capítulo 8. Aquí sólo mostramos una estructura de circuito que cumple los requisitos, pero dejaremos la deducción para dicho capítulo. En la figura 7.24 se observa un contador ascendente de cuatro bits que cuenta en la secuencia 0, 1, 2, . . . ,14, 15, 0, 1 y así sucesivamente. El conteo se indica mediante las salidas del flip-flop $Q_3Q_2Q_1Q_0$. Si suponemos que *Enable* = 1, entonces las entradas *D* de los flip-flops se definen mediante las expresiones

$$D_0 = \overline{Q}_0 = 1 \oplus Q_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

Para un contador más grande la etapa *i*-ésima se define por

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2} \cdots Q_1Q_0$$

En el capítulo 8 mostraremos cómo derivar estas ecuaciones.

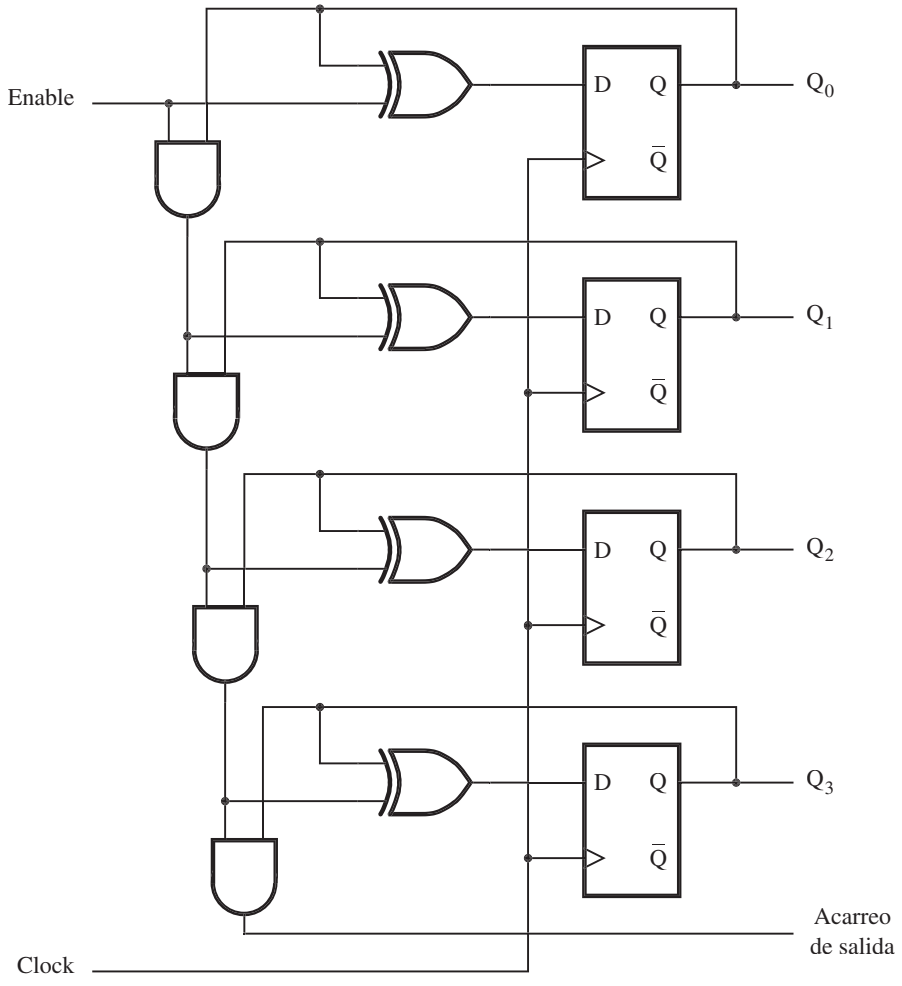


Figura 7.24 Un contador de cuatro bits con flip-flops D.

Hemos incluido la señal de control *Enable* de manera que el contador cuente, valga la expresión, los pulsos del reloj sólo cuando *Enable* = 1. De hecho, las ecuaciones anteriores se modificaron para implementar el circuito de la figura como sigue

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$
$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$$

El funcionamiento del contador se basa en nuestra observación de la tabla 7.1 acerca de que el estado del flip-flop en la etapa *i* cambia sólo si todos los flip-flops anteriores están en el estado Q = 1.

Esto ocasiona que la salida de la compuerta AND que alimenta la etapa *i* sea igual a 1, lo cual hace que la salida de la compuerta XOR conectada a D_i sea igual a $\overline{Q}_i$. De lo contrario, la salida de la compuerta XOR proporciona $D_i = Q_i$, y el flip-flop permanece en el mismo estado. Esto se parece a la propagación de acarreo en un circuito sumador con acarreo de adelanto (véase la sección 5.4); por consiguiente, la cadena de la compuerta AND puede considerarse la *cadena de acarreo*. Aun cuando el circuito es sólo un contador de cuatro bits, hemos incluido un AND adicional que produce el "acarreo de salida". Esta señal facilita la concatenación de dos de estos contadores de cuatro bits para crear un contador de ocho bits.

Finalmente, el lector debe notar que el contador de la figura 7.24 es, en esencia, el mismo que el circuito de la figura 7.23. En la figura 7.16*a* mostramos que un flip-flop T puede formarse a partir de un flip-flop D si se proporcionan las compuertas adicionales que dan

$$\begin{aligned} D &= Q\overline{T} + \overline{Q}T \\ &= Q \oplus T \end{aligned}$$

De esta manera, en cada etapa de la figura 7.24 el flip-flop D y la compuerta XOR asociada implementan la funcionalidad de un flip-flop T.

7.9.3 CONTADORES CON CARGA EN PARALELO

A menudo es necesario empezar a contar a partir de 0. Tal estado puede lograrse usando la capacidad para borrar los flip-flops como se indica en la figura 7.23. Pero a veces es conveniente comenzar con un conteo distinto. Para permitir este modo de operación un circuito contador debe tener algunas entradas a través de las cuales pueda cargarse el conteo inicial. Usar las entradas *Clear* y *Preset* para tal propósito es una posibilidad, mas enseguida se expone un mejor método.

El circuito de la figura 7.24 puede modificarse para brindar la capacidad de carga en paralelo como se muestra en la figura 7.25. Un multiplexor de dos entradas se inserta antes de cada entrada *D*. Una entrada del multiplexor se usa para proporcionar la operación de conteo normal; la otra es un bit de datos que puede cargarse directamente en el flip-flop. Una entrada de control, *Load*, se utiliza para elegir el modo de operación. El circuito cuenta cuando $Load = 0$. Un nuevo valor inicial, $D_3D_2D_1D_0$, se carga en el contador cuando $Load = 1$.

7.10 RESET SÍNCRONO

Ya mencionamos que es importante poder borrar, o *inicializar*, el contenido de un contador antes de empezar una operación de conteo. Ello se logra usando la capacidad de borrado de cada flip-flop. Sin embargo, es posible que también estemos interesados en establecer el conteo en 0 durante el proceso de conteo normal. Un contador ascendente de *n* bits funciona naturalmente como un contador módulo 2^n. Supóngase que queremos tener un contador que cuente en módulo cierta base que no sea una potencia de 2. Por ejemplo, tal vez queramos diseñar un contador módulo 6, para el que la secuencia de conteo sea 0, 1, 2, 3, 4, 5, 0, 1 y así sucesivamente.

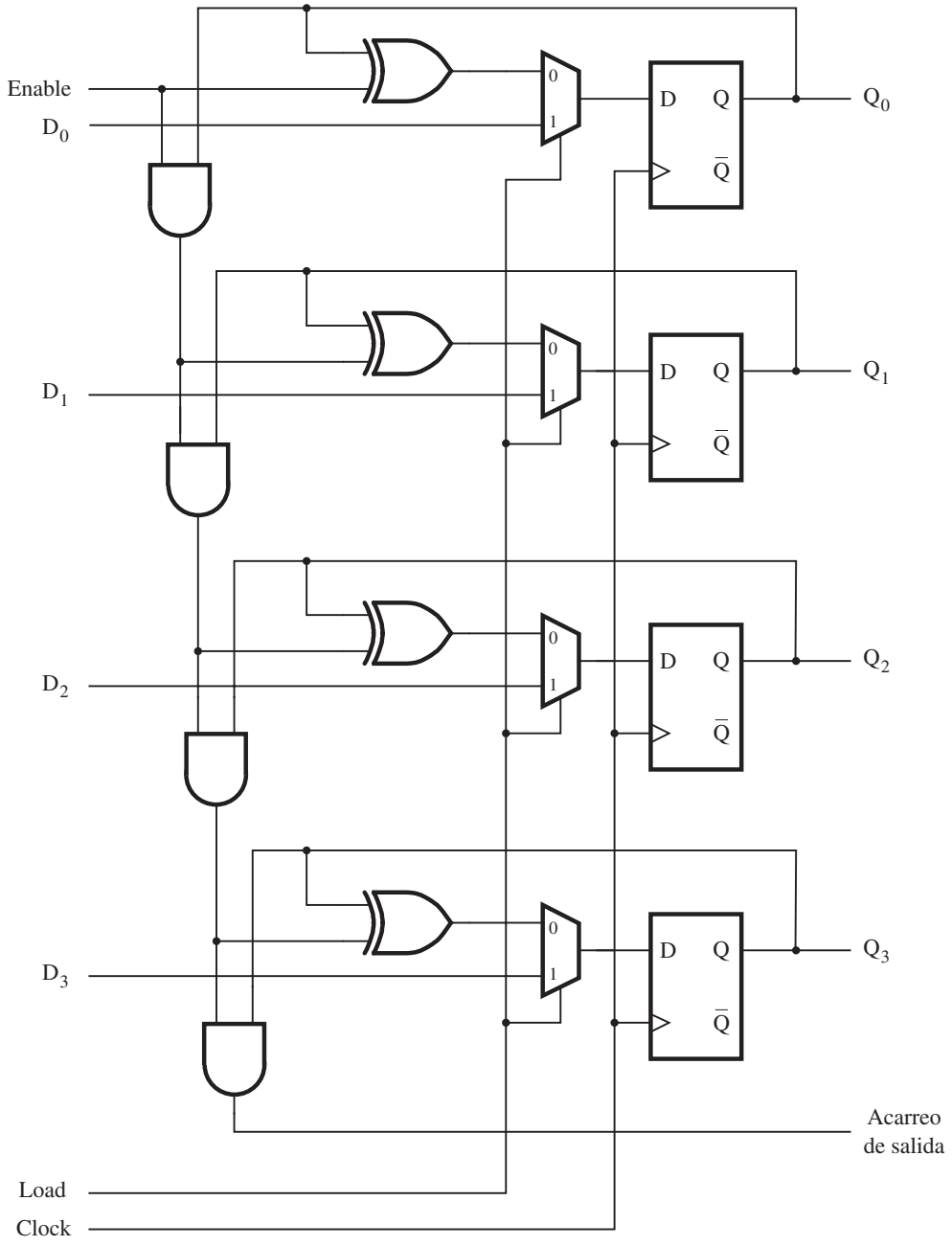
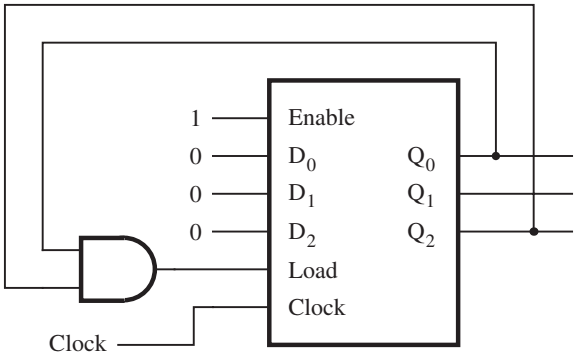
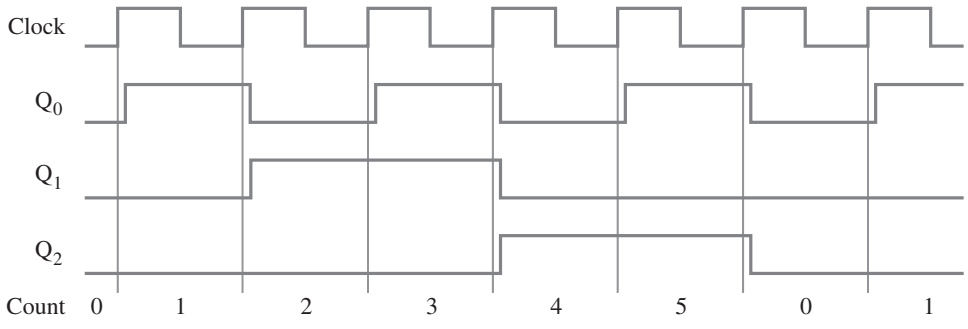


Figura 7.25 Un contador con capacidad de carga en paralelo.

El método más sencillo es reconocer cuándo el conteo llega a 5 y luego inicializar el contador. Puede usarse una compuerta AND para detectar que el conteo alcancó el 5. En realidad, basta determinar que $Q_2 = Q_0 = 1$, lo cual es verdadero sólo para 5 en nuestra secuencia de conteo. En la figura 7.26*a* aparece un circuito basado en este método. Utiliza un contador síncrono de tres bits del tipo representado en la figura 7.25. La función de carga en paralelo del contador se usa para inicializar su contenido cuando el contador llega a 5. La acción de inicialización ocurre en el flanco positivo del reloj después que el conteo alcanza el 5. Comprende la carga de $D_2D_1D_0 = 000$ en los flip-flops. Como se vio en el diagrama de tiempo de la figura 7.26*b*, la secuencia de conteo buscada se logra con cada valor del conteo establecido para un ciclo completo del reloj. Como el contador se inicializa en el flanco activo del reloj, decimos que este tipo de contador tiene un *reset síncrono*.



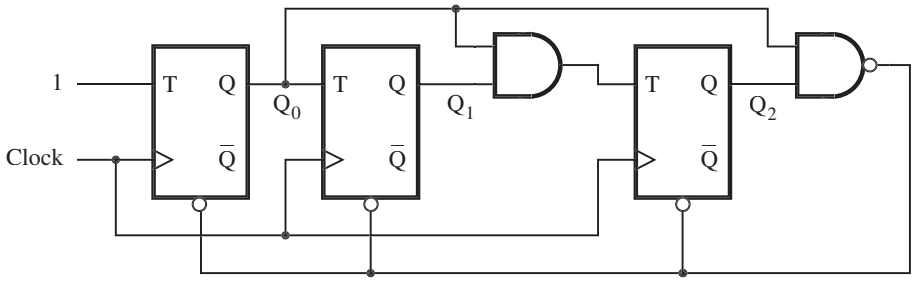
a) Circuito



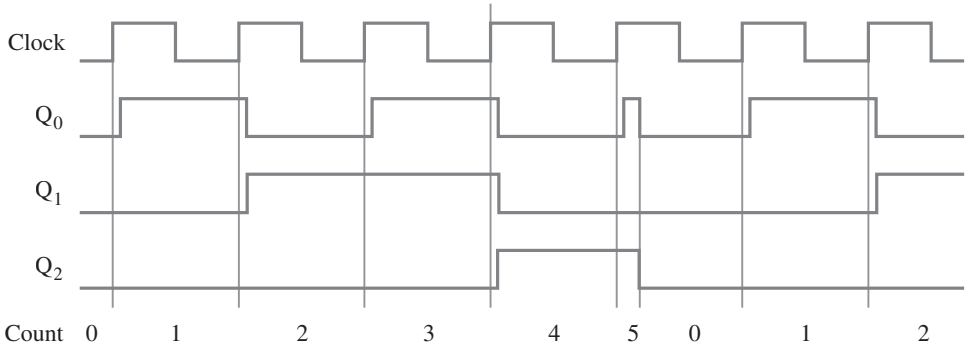
b) Diagrama de tiempo

Figura 7.26 Un contador módulo 6 con reset síncrono.

Considérese ahora la posibilidad de usar la función de borrado de los flip-flops individuales, en vez del método de carga en paralelo. En el circuito de la figura 7.27*a* se ilustra una posibilidad; utiliza la estructura del contador de la figura 7.22*a*. Como las entradas *Clear* están activas en nivel bajo, se utiliza una compuerta NAND para detectar la ocurrencia del conteo de 5 y borrar los tres flip-flops. Desde el punto de vista conceptual, esto parece funcionar bien, pero una revisión más a fondo revela un problema potencial. El diagrama de tiempo para este circuito se presenta en la figura 7.27*b*. Muestra una dificultad que surge cuando el conteo llega a 5. En cuanto el contador llega a su valor, la compuerta NAND desencadena la acción de inicialización. Los flip-flops se borran en 0 poco tiempo después que la compuerta NAND ha detectado el conteo de 5. Este instante depende de los retrasos de compuerta en el circuito, pero no del reloj. Por consiguiente, los valores de señal $Q_2Q_1Q_0 = 101$ se conservan durante un tiempo mucho menor que un ciclo del reloj. Según la aplicación concreta de dicho contador, éste puede ser adecuado pero también completamente inaceptable. Por ejemplo, si el contador se usa en un sistema digital donde todas las operaciones del sistema están sincronizadas por el mismo reloj, entonces este pulso estrecho que indica *Count* = 5 no sería visto por el resto del sistema. Para resolver el problema



a) Circuito



b) Diagrama de tiempo

Figura 7.27 Un contador módulo 6 con un reset asíncrono.

podríamos tratar de usar un contador módulo 7 en su lugar, suponiendo que el sistema ignoraría el pulso corto que indica el conteo de 6. Ésta no es una buena forma de diseñar circuitos porque los pulsos no deseados a menudo causan dificultades imprevistas en la práctica. Se dice que el método empleado en la figura 7.27*a* usa un *reset asíncrono*.

Los diagramas de tiempo de las figuras 7.26*b* y 7.27*b* indican que el reset síncrono es una mejor opción que el asíncrono. La misma observación es cierta si la secuencia de conteo natural ha de dividirse por la carga de un valor distinto de cero. El nuevo valor de conteo puede establecerse limpiamente mediante la función de carga en paralelo. La opción de usar las capacidades de clear y preset de los flip-flops individuales para establecer en 1 sus estados a fin de reflejar el conteo deseado plantea los mismos problemas estudiados junto con el reset asíncrono.

7.11 OTROS TIPOS DE CONTADORES

En esta sección estudiaremos otros tres tipos de contadores que pueden encontrarse en las aplicaciones prácticas. El primero usa la secuencia de conteo decimal y los otros dos generan secuencias de códigos que no representan números binarios.

7.11.1 CONTADOR BCD

Los contadores binario-codificado-decimal (BCD) pueden diseñarse con el método expuesto en la sección 7.10. En la figura 7.28 se muestra un contador de dos dígitos BCD. Consta de dos contadores módulo 10, uno por cada dígito BCD, que implementamos usando el contador de cuatro bits de carga en paralelo de la figura 7.25. Obsérvese que en un contador módulo 10 es preciso inicializar los cuatro flip-flops después de que se ha obtenido el conteo de 9. De esta manera la entrada *Load* para cada etapa es igual a 1 cuando $Q_3 = Q_0 = 1$, lo que ocasiona que los ceros se carguen en los flip-flops en el siguiente flanco positivo de la señal de reloj. Siempre que el conteo en la etapa 0, BCD_0, llega a 9 es necesario habilitar la segunda etapa de modo que se incremente cuando llegue el siguiente pulso del reloj. Ello se logra al mantener la señal *Enable* para BCD_1 baja en todo momento excepto cuando $BCD_0 = 9$.

En la práctica debe ser posible borrar el contenido del contador al activar una señal de control. Dos compuertas OR se incluyen en el circuito para tal fin. La entrada de control *Clear* puede usarse para cargar los ceros en el contador. Nótese que en este caso *Clear* está activa cuando es alta. El código de VHDL para un contador BCD de dos dígitos se da en la figura 7.77.

En cualquier sistema digital suele haber una o más señales de reloj usadas para manejar todo el sistema de circuitos síncrono. En el contador anterior, así como en todos los presentados en las figuras previas, hemos supuesto que el objetivo es contar el número de pulsos del reloj. Desde luego, estos contadores sirven para contar el número de pulsos de cualquier señal que pueda usarse en vez de la de reloj.

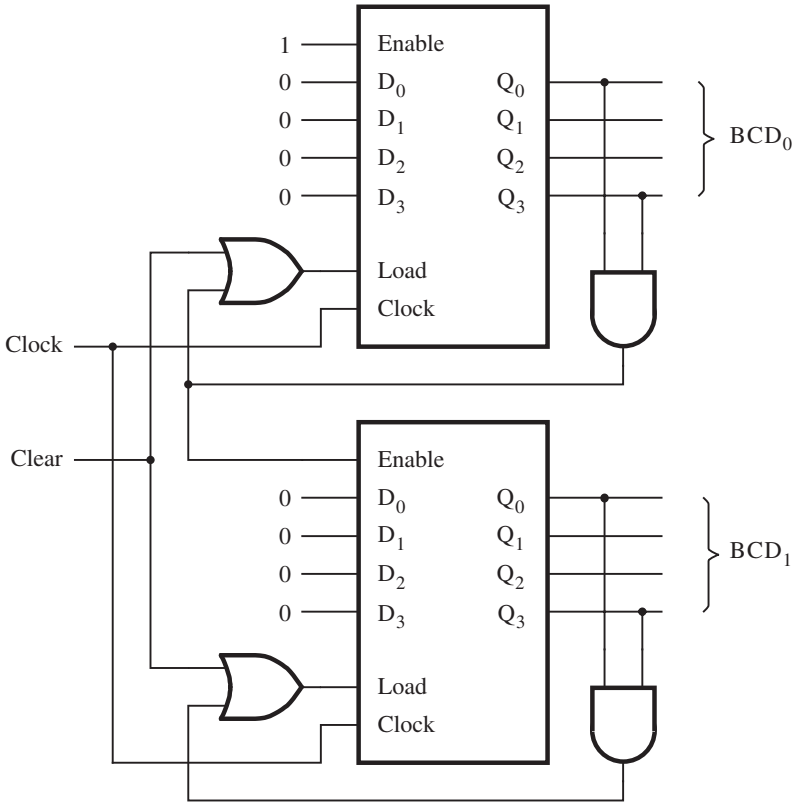
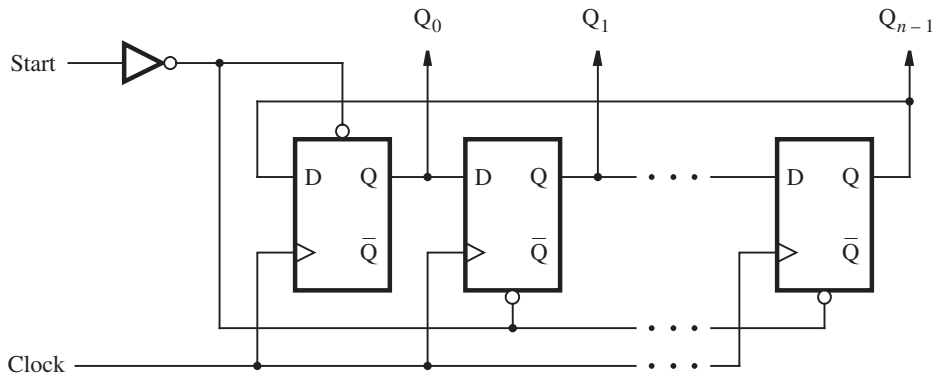


Figura 7.28 Un contador BCD de dos dígitos.

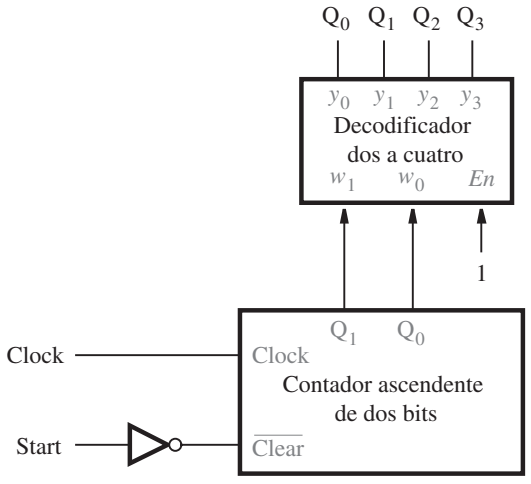
7.11.2 CONTADOR EN ANILLO

En los contadores anteriores el conteo se indica por medio del estado de los flip-flops del contador. En todos los casos el conteo es un número binario. Al usar dichos contadores, si se va a tomar una medida como resultado de un conteo en particular, es necesario detectar la ocurrencia de este conteo. Esto puede hacerse usando compuertas AND, como se ilustra en las figuras 7.26 a 7.28.

Es posible concebir un circuito tipo contador en el que cada flip-flop alcance el estado $Q_i = 1$ para exactamente un conteo, mientras que para todos los demás conteos $Q_i = 0$. Entonces Q_i indica directamente una ocurrencia del conteo correspondiente. En realidad, como esto no representa números binarios, es mejor decir que las salidas de los flips-flops representan un código. Un circuito como éste puede construirse a partir de un registro de corrimiento simple, como se indica en la figura 7.29*a*. La salida Q de la última etapa en el registro de corrimiento se realimenta como la entrada a la primera etapa, la cual crea una estructura en forma de anillo. Si un solo 1 se inyecta en el anillo, este 1 se desplazará a través del anillo en ciclos del reloj sucesivos.



a) Un contador de *n* bits en anillo



b) Un contador de cuatro bits en anillo

Figura 7.29 Contador en anillo.

Por ejemplo, en una estructura de cuatro bits, los códigos posibles $Q_0Q_1Q_2Q_3$ serán 1000, 0100, 0010 y 0001. Como dijimos en la sección 6.2, dicha codificación, donde hay un solo 1 y el resto de las variables de código son 0, se llama *codificación de 1 activo*.

El circuito de la figura 7.29*a* se conoce como *contador en anillo*. Su operación tiene que inicializarse inyectando un 1 en la primera etapa, lo que se logra utilizando la señal de control *Start*, que prestablece en 1 el flip-flop del extremo izquierdo y borra los otros flip-flops hasta dejarlos en 0. Suponemos que todos los cambios en el valor de la señal *Start* ocurren poco tiempo después de un flanco activo del reloj, de manera que los parámetros de tiempo del flip-flop no sean incumplidos.

El circuito de la figura 7.29*a* puede usarse para construir un contador en anillo con cualquier número de bits, *n*. Para el caso específico de $n = 4$, el inciso (*b*) de la figura muestra cómo puede construirse un contador en anillo usando un contador ascendente de dos bits y un decodificador. Cuando *Start* se establece en 1, el contador se restablece en 00. Después que *Start* cambia de nuevo a 0, el contador aumenta su valor de manera normal. El decodificador dos a cuatro, descrito en la sección 6.2, cambia la salida del contador en codificación de 1 activo. Para los valores de conteo 00, 01, 10, 11, 00, etc., el decodificador produce $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$ y sucesivamente. Esta estructura del circuito puede usarse para contadores en anillo más grandes, siempre que el número de bits sea una potencia de dos. En la sección 7.14 daremos un ejemplo de un circuito más grande que emplea el contador en anillo de la figura 7.29*b* como un subcircuito.

7.11.3 CONTADOR JOHNSON

Una variación interesante del contador en anillo se obtiene si, en vez de la salida Q, tomamos la salida $\overline{Q}$ de la última etapa y la retroalimentamos en la primera etapa, como se muestra en la figura 7.30. Este circuito se conoce como *contador Johnson*. Un contador de *n* bits de este tipo genera una secuencia de conteo de longitud 2*n*. Por ejemplo, un contador de cuatro bits produce la secuencia 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, etc. Nótese que en esta secuencia solamente un bit tiene un valor diferente para dos códigos consecutivos.

Para empezar la operación del contador Johnson es preciso inicializar todos los flip-flops, como se muestra en la figura. Obsérvese que ni el contador Johnson ni el contador en anillo generarán la secuencia de conteo buscada si no se inicializa apropiadamente.

7.11.4 COMENTARIOS SOBRE EL DISEÑO DEL CONTADOR

Los circuitos secuenciales presentados en este capítulo, concretamente los registros y contadores, tienen una estructura que permite que los circuitos se diseñen mediante un método intuitivo. En el capítulo 8 presentaremos un enfoque más formal para diseñar circuitos secuenciales y mostrar cómo los circuitos presentados en este capítulo pueden derivarse siguiéndolo.

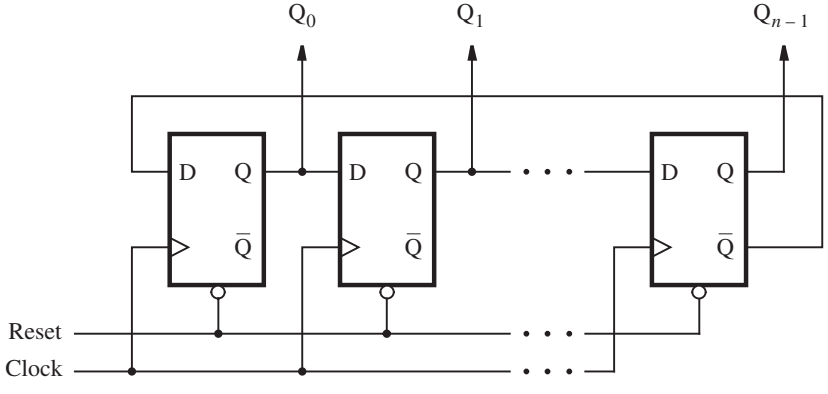


Figura 7.30 Contador Johnson.

7.12 USO DE ELEMENTOS DE ALMACENAMIENTO CON HERRAMIENTAS CAD

En esta sección se muestra cómo diseñar circuitos con elementos de almacenamiento usando ya sea una captura esquemática o código de VHDL.

7.12.1 INCLUSIÓN DE ELEMENTOS DE ALMACENAMIENTO EN ESQUEMAS

Una forma de crear un circuito consiste en dibujar un esquema que construya latches y flip-flops a partir de compuertas lógicas. Como estos elementos de almacenamiento se utilizan en muchas aplicaciones, la mayor parte de los sistemas CAD los proporcionan como módulos preconstruidos. En la figura 7.31 se muestra un esquema creado con una herramienta de captura esquemática; incluye tres tipos de flip-flops que se importan de una biblioteca provista como parte del sistema CAD. El elemento superior es un latch D asíncrono, el de en medio es un flip-flop D disparado por flanco positivo y el inferior es un flip-flop T disparado por flanco positivo. Los flip-flops D y T tienen entradas clear y preset asíncronas, activas en nivel bajo. Si estas entradas no están conectadas en un esquema, entonces la herramienta CAD las vuelve inactivas asignándoles el valor predeterminado de 1.

Cuando el latch D asíncrono se sintetiza para implementarlo en un chip, la herramienta CAD tal vez no genere las compuertas NOR o NAND con acoplamiento cruzado expuestas en la sección 7.2. En algunos chips, como los CPLD, el circuito AND-OR descrito en la figura 7.32 puede ser preferible. Este circuito equivale funcionalmente a la versión con acoplamiento cruzado de la sección 7.2. El circuito de suma de productos se usa porque es más adecuado para

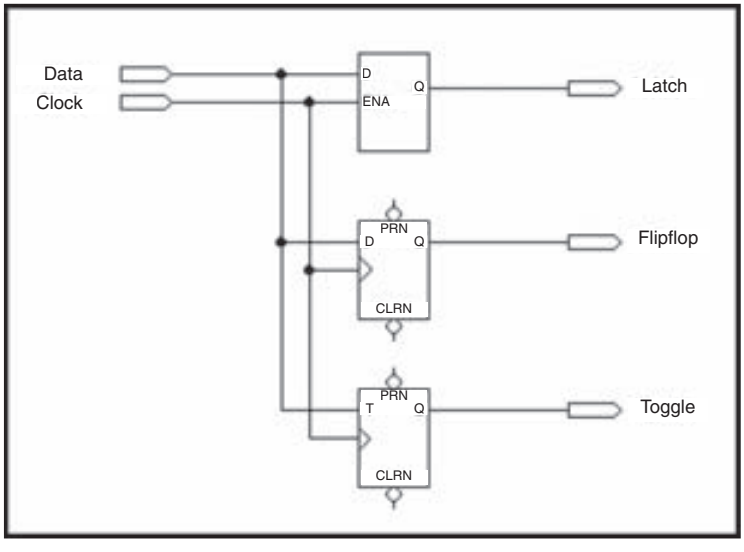


Figura 7.31 Tres tipos de elementos de almacenamiento en un esquema.

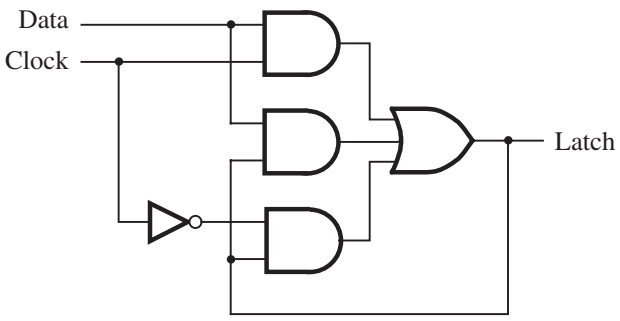


Figura 7.32 Latch D asíncrono generado mediante herramientas CAD.

la implementación en una macrocelda CPLD. Cabe destacar un aspecto de este circuito. Desde el punto de vista funcional, parece que el circuito puede simplificarse si se elimina la compuerta AND con las entradas *Data* y *Latch*. Sin esta compuerta, la compuerta AND superior establece el valor almacenado en el latch cuando el reloj está en 1, y la compuerta AND inferior mantiene el valor almacenado cuando el reloj es 0. Pero sin esta compuerta el circuito tiene un problema de tiempo conocido como *riesgo estático*. En la sección 9.6 explicaremos pormenorizadamente los riesgos.

El circuito de la figura 7.31 puede implementarse en un CPLD como se muestra en la figura 7.33. Los flip-flops D y T se realizan usando en el chip flip-flops configurables como tipos D o T. En la figura se representan en gris oscuro las compuertas y los cables requeridos para implementar el circuito de la figura 7.31.

Los resultados de una simulación de tiempo para la implementación de la figura 7.33 se dan en la figura 7.34. La señal *Latch*, que es la salida del latch D asíncrono, implementado como se indicó en la figura 7.32, sigue a la entrada *Data* siempre que la señal de *Clock* es 1. Debido a los retrasos de propagación en el chip, la señal *Latch* se retarda en el tiempo respecto a la señal *Data*. Como la señal *Flipflop* es la salida del flip-flop D, cambia sólo después de un flanco positivo del reloj. De igual modo, la salida del flip-flop T, llamada *Toggle* en la figura, alterna cuando *Data* = 1 y ocurre un flanco positivo del reloj. El diagrama de tiempo ilustra el retraso cuando el flanco positivo del reloj ocurre en el pin de entrada del chip hasta que un cambio en la salida del flip-flop aparece en el pin de salida del chip. A este tiempo se le llama *tiempo desde reloj hasta la salida*, t_{co}.

7.12.2 USO DE CONSTRUCTORES DE VHDL PARA ELEMENTOS DE ALMACENAMIENTO

En la sección 6.6 describimos una serie de instrucciones de asignación de VHDL. Las instrucciones IF y CASE se presentaron como dos tipos de instrucciones de asignación secuencial. En esta sección mostramos cómo usar tales instrucciones para describir elementos de almacenamiento.

En la figura 6.43, que se repite en la figura 7.35, se brinda un ejemplo de código de VHDL que tiene memoria implícita. Como el código no especifica qué valor debe tener la señal *AeqB* cuando la condición para la instrucción IF no está satisfecha, la semántica especifica que en este

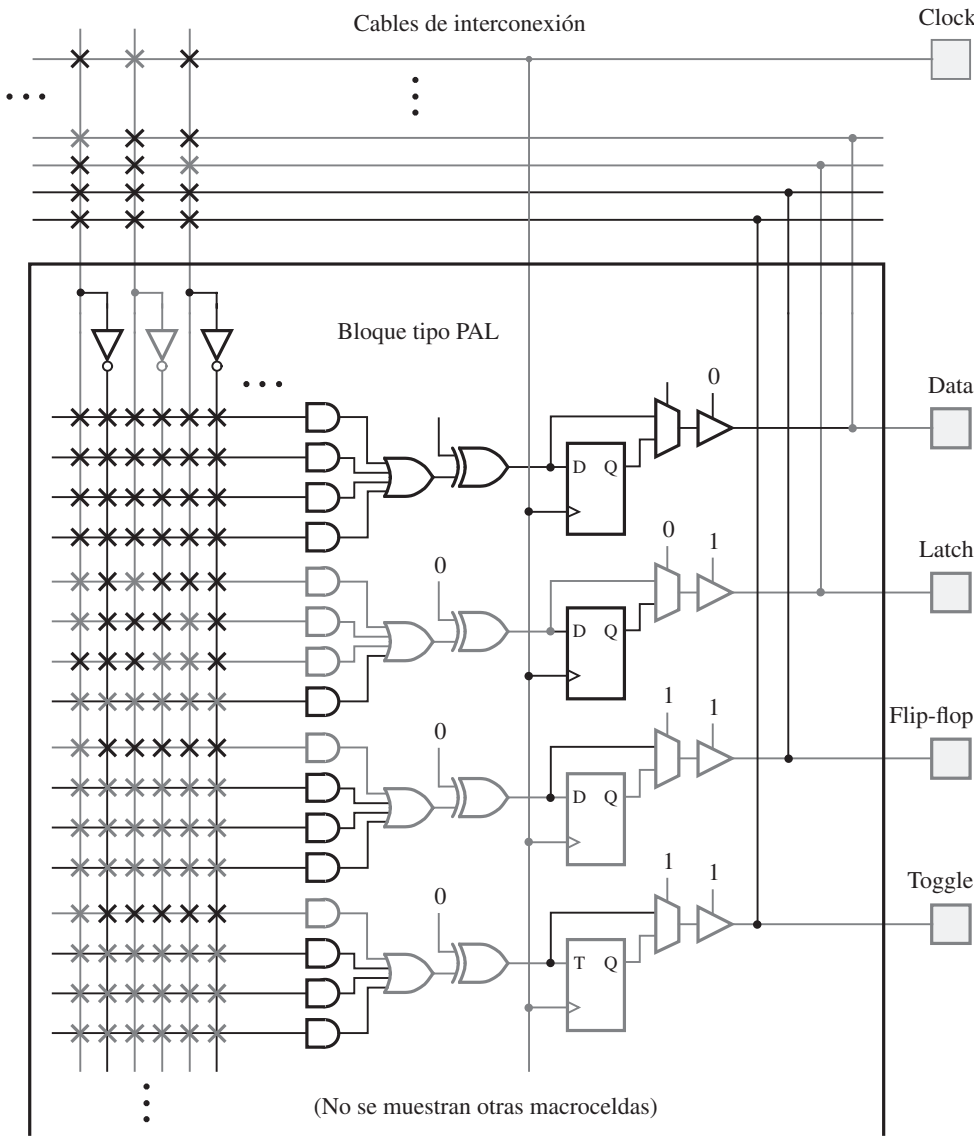


Figura 7.33 Implementación del esquema de la figura 7.31 en un CPLD.

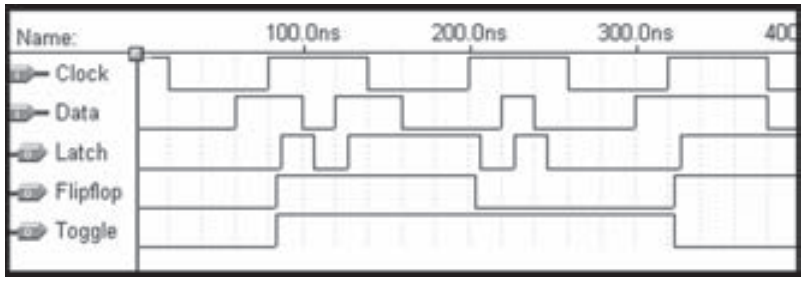


Figura 7.34 Simulación de tiempo para los elementos de almacenamiento de la figura 7.31.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B   : IN     STD_LOGIC ;
           AeqB : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.35 El código de la figura 6.43, ilustrando la memoria implícita.

caso *AeqB* debe conservar su valor actual. La memoria implícita es el concepto clave usado para describir los elementos de circuitos secuenciales, los cuales ilustraremos con varios ejemplos.

Ejemplo 7.1

CÓDIGO PARA UN LATCH D ASÍNCRONO El código de la figura 7.36 define una entidad llamada *latch*, la cual tiene las entradas *D* y *Clk* y la salida Q. El proceso utiliza una instrucción if-then-else para definir el valor de la salida Q. Cuando *Clk* = 1, Q toma el valor de *D*. Para el caso en que *Clk* no es 1, el código no especifica qué valor debe tener Q. Por consiguiente, Q conservará su valor actual en este caso, y el código describe un latch D asíncrono. La lista de

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, Clk  : IN    STD_LOGIC ;
           Q       : OUT   STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, Clk )
    BEGIN
        IF Clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.36 Código para un latch D asíncrono.

sensibilidad del proceso incluye tanto *Clk* como *D* porque estas señales pueden causar un cambio en el valor de la salida Q.

Ejemplo 7.2 **CÓDIGO PARA UN FLIP-FLOP D** En la figura 7.37 se define una entidad llamada *flipflop*, que es un flip-flop D disparado por flanco positivo. El código es idéntico al de la figura 7.36 con dos excepciones. Primera, la lista de sensibilidad del proceso contiene sólo la señal de reloj que puede causar un cambio en la salida Q. Segunda, la instrucción if-then-else utiliza una condición diferente de la empleada en el latch. La sintaxis Clock'EVENT usa un constructor de VHDL llamado *atributo*. Un atributo se refiere a la propiedad de un objeto, como una señal. En este caso el atributo 'EVENT se refiere a cualquier cambio en la señal *Clock*. Combinar la condición Clock'EVENT con la condición *Clock* = 1 significa que "el valor de la señal *Clock* acaba de cambiar, y el valor ahora es igual a 1". Por consiguiente, la condición se refiere a un flanco positivo del reloj. Puesto que la salida Q cambia sólo como resultado de un flanco positivo del reloj, el código describe un flip-flop D disparado por flanco positivo.

Ejemplo 7.3 **CÓDIGO OPCIONAL PARA UN FLIP-FLOP D** El proceso de la figura 7.38 emplea una sintaxis distinta de aquella de la figura 7.37 para describir el flip-flop D. Utiliza la instrucción WAIT UNTIL Clock'EVENT AND Clock = '1', que produce el mismo efecto que la instrucción IF de la figura 7.37. Un proceso que usa una instrucción WAIT UNTIL es un caso especial porque se omite la lista de sensibilidad. El constructor WAIT UNTIL implica que dicha lista incluye sólo la señal de reloj. En nuestro uso de VHDL, el cual es para la síntesis de circuitos, un proceso puede utilizar una instrucción WAIT UNTIL sólo si es su primera instrucción.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.37 Código para un flip-flop D.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figura 7.38 Código equivalente al de la figura 7.37, que usa una instrucción WAIT UNTIL.

En realidad, el atributo 'EVENT es redundante en la instrucción WAIT UNTIL. Podemos simplemente escribir

```
WAIT UNTIL Clock = '1';
```

lo cual también implica que la acción ocurre cuando la señal *Clock* se vuelve igual a 1, es decir, en el flanco donde la señal cambia de 0 a 1. Sin embargo, algunas herramientas CAD de síntesis requieren la inclusión del atributo 'EVENT, razón por la que usamos este estilo en el libro.

En general, siempre que se quiere incluir en el código de VHDL flip-flops sincronizados por el flanco positivo del reloj se usa la condición Clock'EVENT AND Clock '1'. Cuando esta condición aparece en una instrucción IF, cualesquiera señales a las que se asignen valores dentro de la instrucción IF se implementan como las salidas de los flip-flops. Cuando la condición se usa en una instrucción WAIT UNTIL, cualquier señal a la que se asigna un valor en todo el proceso se implementa como la salida de un flip-flop.

Las diferencias entre usar instrucciones IF e instrucciones WAIT UNTIL se estudian con más detalle en el apéndice A, sección A.10.3.

Ejemplo 7.4 **BORRADO ASÍNCRONO** En la figura 7.39 se muestra un proceso parecido al de la figura 7.37. Describe un flip-flop D con una entrada de reset asíncrono activo en nivel bajo (clear). Cuando *Resetn*, la entrada de inicialización, es igual a 0, la salida Q del flip-flop se establece en 0.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN     STD_LOGIC ;
           Q                 : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.39 Flip-flop D con reset asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.40 Flip-flop D con reset síncrono.

Ejemplo 7.5

BORRADO SÍNCRONO En la figura 7.40 se muestra cómo describir un flip-flop D con una entrada de reset síncrono. En este caso la señal reset sólo actúa cuando llega un flanco positivo del reloj. El código genera el circuito de la figura 7.15, el cual tiene una compuerta AND conectada a la entrada D del flip-flop.

En la figura A.33*a* del apéndice A se muestra cómo el mismo circuito se especifica usando una instrucción IF en vez de WAIT UNTIL.

7.13 USO DE REGISTROS Y CONTADORES CON HERRAMIENTAS CAD

En esta sección mostraremos cómo incluir registros y contadores en circuitos diseñados con herramientas CAD. En los ejemplos expuestos se usa captura esquemática y código de VHDL.

7.13.1 INCLUSIÓN DE REGISTROS Y CONTADORES EN ESQUEMAS

En la sección 5.5.1 explicamos que un sistema CAD suele incluir bibliotecas de subcircuitos integrados. Presentamos la biblioteca de módulos parametrizados (LPM) y usamos el módulo su-

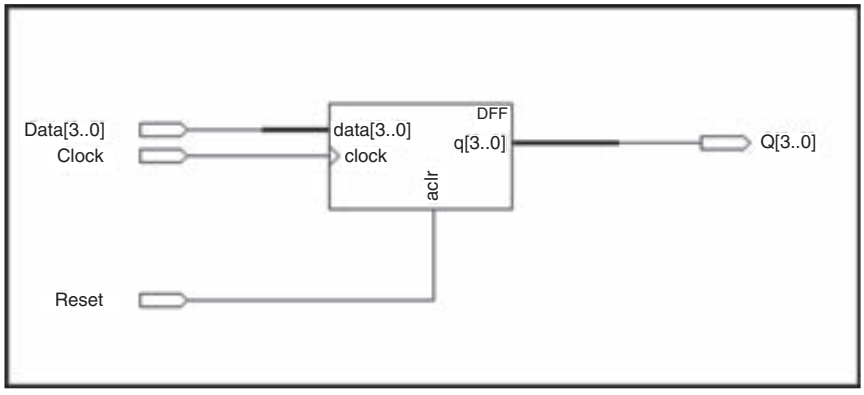


Figura 7.41 El módulo de flip-flop parametrizado *lpm_ff*.

mador/restador *lpm_add_sub* como ejemplo. La LPM incluye subcircuitos que tienen módulos que constituyen los flip-flops, registros, contadores y muchos otros circuitos útiles. En la figura 7.41 se muestra un símbolo que representa el módulo *lpm_ff*. Este módulo es un registro con uno o más flip-flops disparados por flanco positivo que pueden ser de tipo D o T. El módulo tiene parámetros que permiten elegir el número y tipo de flip-flops. En este caso decidimos tener cuatro flip-flops D. El tutorial del apéndice C explica cómo se realiza la configuración del módulo.

Las entradas *D* a los cuatro flip-flops, llamadas *data* en el símbolo gráfico, están conectadas a las cuatro señales de entrada de cuatro bits *Data*[3..0]. La entrada reset asíncrono (borrado) activa en nivel alto, *aclr*, se muestra en el esquema. Las salidas del flip-flop, *q*, están conectadas al símbolo de salida etiquetado como Q[3..0].

En la sección 7.3 dijimos que una aplicación útil de los flip-flops D consiste en almacenar los resultados de un cálculo aritmético, como la salida de un circuito sumador. Un ejemplo se muestra en la figura 7.42, donde se utilizan dos módulos de la LPM, *lpm_add_sub* y *lpm_ff*. El módulo *lpm_add_sub* se describió en la sección 5.5.1. Sus parámetros, no mostrados en la figura 7.42, están preparados para configurar el módulo como un circuito sumador de cuatro bits. La entrada de datos de cuatro bits *data* del sumador está manejada por la señal de entrada *Data*[3..0]. Los bits de suma, *result*, están conectados a las entradas de datos del *lpm_ff*, el cual está configurado como un registro *D* de cuatro bits con borrado asíncrono. El registro genera la salida del circuito, Q[3..0], que aparece en el lado izquierdo del esquema. Esta señal se realimenta hacia la entrada *datab* del sumador. Los bits de suma del sumador también se proporcionan como una salida del circuito, *Sum*[3..0], para facilidad de referencia en el análisis siguiente. Si el registro se borra primero y se establece en 0000, entonces el circuito puede emplearse para sumar los números binarios de la entrada *Data*[3..0] a una suma que está siendo acumulada en el registro, si un número nuevo se aplica a la entrada durante cada ciclo del reloj. El circuito que realiza esta función recibe el nombre de circuito *acumulador*.

Sintetizamos un circuito a partir del esquema e implementamos el sumador de cuatro bits usando la estructura de acarreo de adelanto. En la figura 7.43 aparece una simulación de tiempo para el circuito. Después de inicializar el circuito, la entrada *Data* se establece en 0001. El

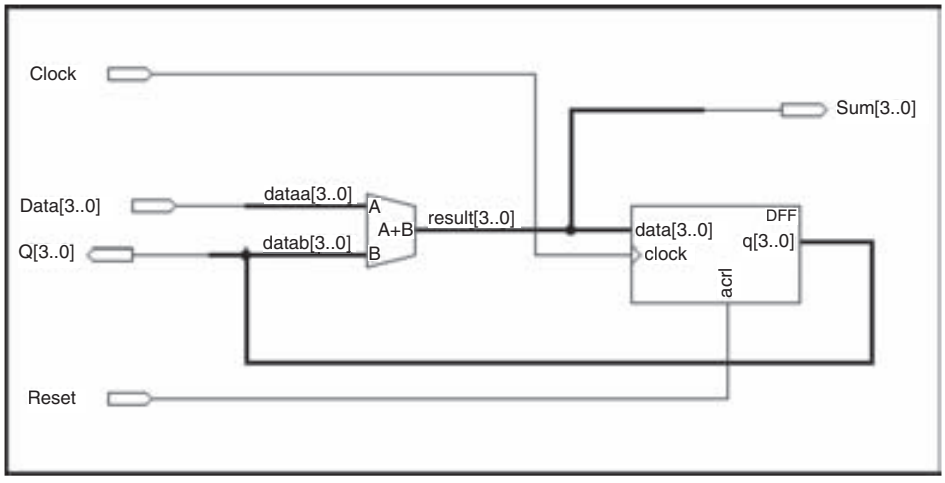


Figura 7.42 Un sumador con retroalimentación registrada.

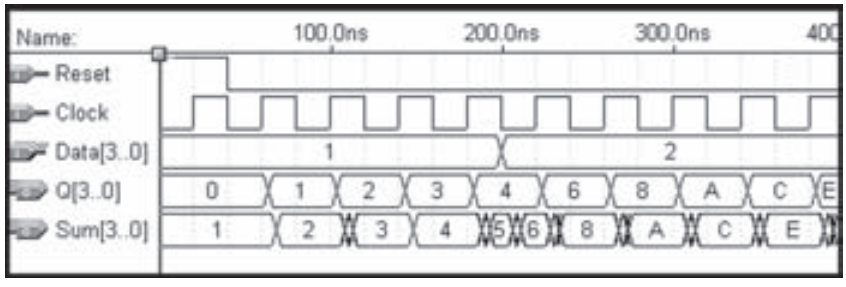


Figura 7.43 Simulación de tiempo del circuito de la figura 7.42.

sumador produce la suma 0000 + 0001 = 0001, la cual luego se sincroniza con el registro en el tiempo 60 ns. Después del retraso t_{co}, Q[3..0] se vuelve 0001, y esto hace que el sumador produzca la nueva suma 0001 + 0001 = 0010. El tiempo necesario para generar la suma nueva está determinado por la velocidad del circuito sumador, el cual produce la suma después de 12.5 ns en este caso. La suma nueva no aparece en la salida Q hasta después del siguiente flanco positivo del reloj, en 100 ns. El sumador produce después 0011 como la suma siguiente. Cuando *Sum* cambia de 0010 a 0011, aparecen algunas oscilaciones en el diagrama de tiempo, ocasionadas por la propagación de las señales de acarreo a través del circuito sumador. Estas oscilaciones no se ven en la salida Q, ya que *Sum* es estable en el momento que ocurre el siguiente flanco positivo del reloj. Al avanzar al tiempo 180 ns, *Sum* = 0100, y este valor se marca en el registro. El sumador produce la nueva suma 0101. Entonces, a los 200 ns, *Data* cambia a 0010, lo que causa que la

suma cambie a 0100 + 0010 = 0110. En el siguiente flanco positivo del reloj, Q se establece en 0110; el valor *Sum* = 0101 que estuvo presente temporalmente en el circuito no se observó en la salida Q. El circuito continúa sumando 0010 a la salida Q en cada flanco positivo del reloj.

Tras simular el comportamiento del circuito, debemos considerar si es posible concluir o no con cierta seguridad que el circuito funciona de manera adecuada. Idealmente, es prudente probar todas las combinaciones posibles de las entradas de un circuito antes de declarar que funciona como se desea. No obstante, en la práctica estas pruebas con frecuencia no son factibles debido al número de combinaciones de entrada que existen. Para el circuito de la figura 7.42 podríamos verificar que el sumador produjo una suma correcta y también que cada uno de los cuatro flip-flops en el registro almacenan ya sea 0 o 1 en forma apropiada. Comentaremos los aspectos relativos a la prueba de los circuitos en el capítulo 11.

Para que el circuito de la figura 7.42 trabaje bien deben cumplirse las restricciones de tiempo siguientes. Cuando el registro marque un flanco positivo del reloj, ha de propagarse un cambio en el valor de la señal en la salida del registro a través de la trayectoria de retroalimentación a la entrada *datab* del sumador. El sumador entonces produce una suma nueva, la cual debe propagarse a la entrada *data* del registro. Para el chip utilizado a fin de implementar el circuito, el retraso total incurrido es de 14 ns. El retraso puede dividirse como sigue: demora 2 ns desde que el registro se sincroniza hasta que un cambio en su salida llega a la entrada *datab* del sumador. El sumador produce una suma nueva en 8 ns, y requiere 4 ns para que se propague a la entrada *data* del registro. En la figura 7.43 el periodo del reloj es 40 ns. Por consiguiente, una vez que una suma nueva llega a la entrada *data* del registro, quedan 40 − 14 = 26 ns hasta que ocurra el siguiente flanco positivo del reloj. La entrada *data* debe estar estable para la cantidad de tiempo de preparación, t_{su} = 3 ns, antes del flanco del reloj. En consecuencia, tenemos 26 − 3 = 23 ns libres. El periodo del reloj puede disminuirse hasta 23 ns y el circuito aún seguirá trabajando. Pero si el periodo del reloj es menor que 40 − 23 = 17 ns, entonces el circuito no funcionará bien. Desde luego, si se usa un chip distinto para implementar el circuito, entonces se producirían resultados de sincronización diferentes. Los sistemas CAD proporcionan herramientas que pueden determinar automáticamente el periodo del reloj mínimo permitido durante el cual un circuito funcionará de manera correcta. El tutorial del apéndice C muestra cómo hacer esto usando las herramientas que acompañan al libro.

7.13.2 REGISTROS Y CONTADORES EN CÓDIGO DE VHDL

Los subcircuitos predefinidos en la biblioteca LPM pueden instanciarse en código de VHDL. En la figura 7.44 se instancia el módulo *lpm_shiftreg*, el cual es un registro de corrimiento de *n* bits. Los parámetros del módulo se fijan usando el constructor GENERIC MAP, como se muestra. El constructor GENERIC MAP es similar al constructor PORT MAP que se emplea para asignar nombres de señal a los puertos de un subcircuito. GENERIC MAP se utiliza para asignar valores a los parámetros del subcircuito. El número de flip-flops en el registro de corrimiento se establece en cuatro usando el parámetro LPM_WIDTH => 4. El módulo puede configurarse para desplazarse a la izquierda o a la derecha. El parámetro LPM_DIRECTION => RIGHT establece que la dirección de desplazamiento será de izquierda a derecha. El código utiliza la entrada de borrado asíncrona activa en nivel alto, *aclr*, del módulo, y la entrada de carga en paralelo activa en nivel alto *load*, la cual permite que el registro de corrimiento se cargue con los datos en paralelo en la entrada *data* del módulo. Cuando el corrimiento se lleva a cabo, el valor de la entrada

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY shift IS
    PORT ( Clock          : IN   STD_LOGIC ;
           Reset          : IN   STD_LOGIC ;
           Shiftin, Load  : IN   STD_LOGIC ;
           R              : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Q              : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift ;

ARCHITECTURE Structure OF shift IS
BEGIN
    instance: lpm_shiftreg
        GENERIC MAP (LPM_WIDTH => 4, LPM_DIRECTION => "RIGHT")
        PORT MAP (data => R, clock => Clock, aclr => Reset,
            load => Load, shiftin => Shiftin, q => Q ) ;
END Structure ;
```

Figura 7.44 Instanciación del módulo *lpm_shiftreg*.

shiftin se desplaza en el flip-flop del extremo izquierdo y el bit corrido hacia afuera aparece en el bit del extremo derecho de la salida *q* en paralelo. El código usa la asociación mencionada, descrita en la sección 5.5.2, para conectar las señales de entrada y salida de la entidad *shift* a los puertos del módulo. Por ejemplo, la señal de entrada *R* se conecta al puerto *data* del módulo. Cuando se traslada a un circuito, el *lpm_shiftreg* tiene la estructura que se exhibe en la figura 7.19.

También hay módulos predefinidos para varios tipos de contadores, los cuales se necesitan comúnmente en los circuitos lógicos. Un ejemplo es el módulo *lpm_counter*, el cual es un contador de ancho variable con entradas de carga en paralelo.

7.13.3 USO DE INSTRUCCIONES SECUENCIALES DE VHDL PARA REGISTROS Y CONTADORES

En vez de instanciar los subcircuitos predefinidos para los registros, los registros de corrimiento, los contadores, etc., los circuitos pueden describirse en VHDL con instrucciones secuenciales. En la figura 7.39 se presenta el código para un flip-flop D. Una manera directa de describir un registro de *n* bits es mediante código jerárquico que incluya *n* instancias del subcircuito del flip-flop D. Un enfoque más simple se muestra en la figura 7.45. Utiliza el mismo código de la figura 7.39, excepto que la entrada D y la salida Q se definen como señales multibit. El código representa un registro de ocho bits con borrado asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "00000000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.45 Código para un registro de ocho bits con borrado asíncrono.

Ejemplo 7.6 **UN REGISTRO DE *N* BITS** Como los circuitos lógicos a menudo necesitan registros de diferentes tamaños, es conveniente definir una entidad de registro para la que el número de flip-flops pueda cambiar fácilmente. En la figura 7.46 se muestra cómo puede extenderse el código de la figura 7.45 para incluir un parámetro que establezca el número de flip-flops. El parámetro es un entero, *N*, que se define usando el constructor de VHDL llamado GENERIC. El valor de *N* se establece en 16 usando el operador de asignación :=. Al cambiar este parámetro, el código puede representar un registro de cualquier tamaño. Si el registro se declara como un componente, entonces sirve como subcircuito en otro código. Ese código puede usar el valor predeterminado para el parámetro GENERIC o especificar de alguna otra forma otro parámetro con el constructor GENERIC MAP. Un ejemplo que muestra cómo se usa GENERIC MAP aparece en la figura 7.44.

Las señales *D* y Q de la figura 7.46 están definidas en términos de *N*. La instrucción que inicializa todos los bits de Q en 0 usa la extraña sintaxis Q <= (OTHERS => '0'). Para el valor predeterminado de $N = 16$, esta instrucción equivale a la instrucción Q <= "0000000000000000". La sintaxis (OTHERS => '0') tiene como resultado un dígito '0' que se asigna a cada uno de los bits de Q, independientemente de cuántos bits tenga Q. Permite que el código se utilice para cualquier valor de *N*, en vez de sólo para $N = 16$.

Ejemplo 7.7 **UN REGISTRO DE CORRIMIENTO DE CUATRO BITS** Suponga que queremos escribir código de VHDL que represente el registro de corrimiento de cuatro bits de la figura 7.19. Un enfoque consiste en escribir código jerárquico que use cuatro subcircuitos. Cada subcircuito se compone de un flip-flop D con un multiplexor dos a uno conectado a la entrada *D*. En la figura 7.47 se define la entidad llamada *muxdff*, que representa este subcircuito. Los dos datos de en-

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.46 Código para un registro de *n* bits con borrado asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY muxdff IS
    PORT ( D0, D1, Sel, Clock  : IN   STD_LOGIC ;
           Q                   : OUT  STD_LOGIC ) ;
END muxdff ;

ARCHITECTURE Behavior OF muxdff IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Sel = '0' THEN
            Q <= D0 ;
        ELSE
            Q <= D1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.47 Código para un flip-flop D con un multiplexor dos a uno en la entrada *D*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT ( R            : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           L, w, Clock  : IN     STD_LOGIC ;
           Q            : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;

ARCHITECTURE Structure OF shift4 IS
    COMPONENT muxdff
        PORT ( D0, D1, Sel, Clock  : IN  STD_LOGIC ;
               Q                   : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    Stage3: muxdff PORT MAP ( w, R(3), L, Clock, Q(3) ) ;
    Stage2: muxdff PORT MAP ( Q(3), R(2), L, Clock, Q(2) ) ;
    Stage1: muxdff PORT MAP ( Q(2), R(1), L, Clock, Q(1) ) ;
    Stage0: muxdff PORT MAP ( Q(1), R(0), L, Clock, Q(0) ) ;
END Structure ;
```

Figura 7.48 Código jerárquico para un registro de corrimiento de cuatro bits.

trada se llaman D_0 y D_1, y se seleccionan por medio de la entrada *Sel*. La instrucción del proceso (process) especifica que si $Sel = 0$ en el flanco positivo del reloj, entonces el valor de D_0 se asigna a Q; de lo contrario, se le asigna el valor de D_1.

En la figura 7.48 se define el registro de corrimiento de cuatro bits. La instrucción etiquetada *Stage*3 instancia el flip-flop del extremo izquierdo, el cual tiene la salida Q_3, y la instrucción *Stage*0 instancia el flip-flop del extremo derecho, Q_0. Cuando $L = 1$, se carga en paralelo desde la entrada *R*, y cuando $L = 0$, el corrimiento se realiza de izquierda a derecha. Los datos seriales se corren hacia el bit más importante, Q_3, desde la entrada *w*.

Ejemplo 7.8 **CÓDIGO OPCIONAL PARA UN REGISTRO DE CORRIMIENTO DE CUATRO BITS** Un estilo diferente de código para el registro de corrimiento de cuatro bits se muestra en la figura 7.49. Las líneas de código están numeradas para que sea más fácil remitirse a ellas. En vez de utilizar subcircuitos, el registro de corrimiento se describe mediante instrucciones secuenciales. Por la instrucción WAIT UNTIL de la línea 13, cualquier señal a la que se asigne un valor dentro del proceso debe implementarse como la salida de un flip-flop. Las líneas 14 y 15 especifican la carga en paralelo del registro de corrimiento cuando $L = 1$. La cláusula ELSE en las líneas 16 a 20 especifica la operación de corrimiento. La línea 17 desplaza el valor de Q_1 hacia el flip-flop con la salida Q_0. Las líneas 18 y 19 desplazan los valores de Q_2 y Q_3 hacia los flip-flops con las salidas Q_1 y Q_2, respectivamente. Por último, la línea 20 desplaza el valor de *w* hacia el flip-flop en el extremo izquierdo, el cual tiene la salida Q_3. Observe que la semántica del proceso, descrita en la sección 6.6.6, estipula que las cuatro tareas de las líneas 17 a 20 están programadas para ocurrir sólo después de que todas las instrucciones del proceso se han evaluado. Por consiguiente, los cuatro flip-flops cambian sus valores al mismo tiempo, según se requiere en el registro de corrimiento. El código genera el mismo circuito de registro de corrimiento que el de la figura 7.48.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R       : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock   : IN      STD_LOGIC ;
6              L, w    : IN      STD_LOGIC ;
7              Q       : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(0) <= Q(1) ;
18              Q(1) <= Q(2);
19              Q(2) <= Q(3) ;
20              Q(3) <= w ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figura 7.49 Código opcional para un registro de corrimiento.

Es importante considerar el efecto de invertir el orden de las líneas 17 a 20 de la figura 7.49, como se indica en la figura 7.50. En este caso la primera operación de corrimiento especificada en el código, en la línea 17, desplaza el valor de *w* hacia el flip-flop del extremo izquierdo con la salida Q_3. Debido a la semántica de la instrucción de proceso (process), la asignación a Q_3 no surte efecto hasta que se evalúan todas las instrucciones subsiguientes dentro del proceso. Por consiguiente, la línea 18 desplaza el valor presente de Q_3 antes de que cambie como resultado de la línea 17, hacia el flip-flop con la salida Q_2. De forma similar, las líneas 19 y 20 desplazan los valores presentes de Q_2 y Q_1 hacia los flip-flops con las salidas Q_1 y Q_0, respectivamente. El código produce el mismo circuito generado con el ordenamiento de las instrucciones de la figura 7.49.0

Ejemplo 7.9

REGISTRO DE CORRIMIENTO DE *N* BITS En la figura 7.51 se muestra el código que puede utilizarse para representar registros de corrimiento de cualquier tamaño. El parámetro *N* de GENERIC, que tiene el valor predeterminado de 8 en la figura, establece el número de flip-flops. El código es idéntico al de la figura 7.49 con dos excepciones. Primera, *R* y Q están definidas en términos de *N*. Segunda, la cláusula ELSE que describe la operación de corrimiento está generalizada para funcionar con cualquier número de flip-flops.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R       : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock   : IN     STD_LOGIC ;
6              L, w    : IN     STD_LOGIC ;
7              Q       : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(3) <= w ;
18              Q(2) <= Q(3) ;
19              Q(1) <= Q(2);
20              Q(0) <= Q(1) ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

Figura 7.50 Código que invierte el orden de las instrucciones de la figura 7.49.

Las líneas 18 a 20 especifican la operación de corrimiento para los flip-flops $N - 1$ del extremo derecho, los cuales tienen las salidas Q_{N-2} a Q_0. El constructor usado se llama FOR LOOP. Es parecido a la instrucción FOR GENERATE presentada en la sección 6.6.4, que sirve para generar un conjunto de instrucciones concurrentes. FOR LOOP se utiliza para producir una serie de instrucciones secuenciales. La primera iteración del ciclo desplaza el valor presente de Q_1 hacia el flip-flop con la salida Q_0. La siguiente iteración desplaza Q_2 hacia el flip-flop con la salida Q_1, y así por el estilo, con la iteración final desplazando Q_{N-1} hacia el flip-flop con la salida Q_{N-2}. La línea 21 completa la operación de corrimiento al desplazar el valor de la entrada serial *w* hacia el flip-flop del extremo izquierdo con la salida Q_{N-1}.

Ejemplo 7.10 **CONTADOR ASCENDENTE** En la figura 7.52 se muestra el código para un contador ascendente de cuatro bits que tiene una entrada de inicialización, *Resetn*, y una entrada de habilitación, *E*. En el cuerpo de arquitectura los flip-flops del contador se representan mediante la señal llamada *Count*. La instrucción del proceso (process) especifica un reset asíncrono de *Count* cuando *Resetn* = 0. La cláusula ELSIF especifica que en el flanco positivo del reloj, si $E = 1$,

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shiftn IS
4       GENERIC ( N : INTEGER := 8 ) ;
5       PORT ( R       : IN      STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
6              Clock   : IN      STD_LOGIC ;
7              L, w    : IN      STD_LOGIC ;
8              Q       : BUFFER  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
9   END shiftn ;

10  ARCHITECTURE Behavior OF shiftn IS
11  BEGIN
12      PROCESS
13      BEGIN
14          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
15          IF L = '1' THEN
16              Q <= R ;
17          ELSE
18              Genbits: FOR i IN 0 TO N-2 LOOP
19                  Q(i) <= Q(i + 1) ;
20              END LOOP ;
21              Q(N-1) <= w ;
22          END IF ;
23      END PROCESS ;
24  END Behavior ;
```

Figura 7.51 Código para un registro de corrimiento de *n* bits, de izquierda a derecha.

el conteo se incrementa. Si $E = 0$, el código asigna explícitamente Count <= *Count*. Esta instrucción no se requiere para describir correctamente el contador debido a la semántica de la memoria implícita, pero podría incluirse en aras de la claridad. Las salidas Q se asignan al valor de *Count* al final del código. El código produce el circuito mostrado en la figura 7.23 si el compilador de VHDL opta por usar flip-flops T, y genera el circuito de la figura 7.24 (con la entrada reset añadida) si el compilador elige flip-flops D.

Ejemplo 7.11

USO DE SEÑALES INTEGER (ENTEROS) EN UN CONTADOR Los contadores con frecuencia se definen en VHDL usando el tipo INTEGER, que explicamos en la sección 5.5.4. El código de la figura 7.53 define el contador ascendente que tiene una entrada de carga en paralelo, además de una entrada reset. Los datos en paralelo, *R*, así como la salida del contador, Q, se definen mediante el tipo INTEGER. Puesto que están en el límite de 0 a 15, ambas señales representan cantidades de cuatro bits. En la figura 7.52 la señal *Count* se define para representar

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clock, Resetn, E  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            ELSE
                Count <= Count ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figura 7.52 Código para un contador ascendente de cuatro bits.

los flip-flops del contador. Esta señal no es necesaria si las salidas Q están en modo BUFFER, como se muestra en la figura 7.53. La instrucción if-then-else al principio del proceso incluye el mismo reset asíncrono de la figura 7.53. La cláusula ELSIF especifica que en el flanco positivo del reloj, si $L = 1$, los flip-flops del contador se cargan en paralelo desde las entradas R. Si $L = 0$, el conteo se incrementa.

Ejemplo 7.12 **CONTADOR DESCENDENTE** En la figura 7.54 se muestra el código para un contador descendente llamado *downcnt*. Para facilitar el cambio del conteo inicial, se define como un parámetro GENERIC llamado *modulus*. En el flanco positivo del reloj, si $L = 1$ el contador se carga con los valores *modulus* – 1, y si $L = 0$, el conteo disminuye. El contador también incluye una entrada de habilitación, E. Establecer $E = 1$ permite que el conteo disminuya cuando ocurre un flanco activo del reloj.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
    PORT ( R                : IN     INTEGER RANGE 0 TO 15 ;
           Clock, Resetn, L : IN     STD_LOGIC ;
           Q                : BUFFER INTEGER RANGE 0 TO 15 ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q + 1 ;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

Figura 7.53 Un contador de cuatro bits con carga en paralelo que usa señales INTEGER.

7.14 EJEMPLOS DE DISEÑO

En esta sección se presentan dos ejemplos de sistemas digitales que utilizan algunos bloques de construcción descritos en este capítulo y en el capítulo 6.

7.14.1 ESTRUCTURA DE BUS

Los sistemas digitales suelen contener un conjunto de registros empleados para almacenar datos. En la figura 7.55 se presenta un ejemplo de un sistema que tiene k registros de n bits, $R1$ a Rk. Cada registro se conecta a un conjunto común de n cables, los cuales se ocupan para transferir datos dentro y fuera de los registros. Este conjunto común de cables suele llamarse *bus*. Además de los registros, en un sistema real otros tipos de bloques de circuitos se conectarían al bus. En la figura se muestra cómo n bits de datos pueden colocarse en el bus desde otro bloque de circuito, usando la entrada de control *Extern*. Los datos almacenados en cualquiera de los registros pueden transferirse por el bus a un registro distinto o al bloque de circuitos conectado al bus.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY downcnt IS
    GENERIC ( modulus : INTEGER := 8 ) ;
    PORT ( Clock, L, E  : IN     STD_LOGIC ;
           Q            : OUT  INTEGER RANGE 0 TO modulus−1 ) ;
END downcnt ;

ARCHITECTURE Behavior OF downcnt IS
    SIGNAL Count : INTEGER RANGE 0 TO modulus−1 ;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
        IF L = '1' THEN
            Count <= modulus−1 ;
        ELSE
            IF E = '1' THEN
                Count <= Count−1 ;
            END IF ;
        END IF ;
    END PROCESS;
    Q <= Count ;
END Behavior ;
```

Figura 7.54 Código para un contador descendente.

Resulta esencial asegurar que sólo un bloque de circuito intenta colocar datos en los cables del bus en cierto instante. En la figura 7.55 cada registro está conectado al bus por medio de un buffer triestado de n bits. Se utiliza un circuito de control para asegurar que sólo una de las entradas enable del buffer triestado, $R1_{out}, \ldots, Rk_{out}$, se valide en cualquier momento dado. El circuito de control también produce las señales $R1_{in}, \ldots, Rk_{in}$, las cuales controlan cuándo se cargan los datos en cada registro. En general, el circuito de control podría realizar una serie de funciones, como la transferencia de los datos almacenados en un registro hacia otro y así por el estilo. En la figura 7.55 se muestra una señal de entrada llamada *Function* que instruye el circuito de control para que realice una tarea específica. El circuito de control está sincronizado por una entrada del reloj, que es la misma señal de reloj que controla los registros k.

En la figura 7.56 se presenta una vista más detallada de cómo pueden conectarse a un bus los registros de la figura 7.55. Para mantener la simplicidad se muestran dos registros de dos bits, pero el mismo esquema puede usarse para registros más grandes. Para el registro $R1$, dos buffers triestado habilitados por $R1_{out}$ se usan a fin de conectar la salida de cada flip-flop a un cable en el bus. La entrada D en cada flip-flop está conectada a un multiplexor dos a uno, cuya entrada select está controlada por $R1_{in}$. Si $R1_{in} = 0$, los flip-flops se cargan desde sus salidas Q; por consiguiente, los datos almacenados no cambian. Pero si $R1_{in} = 1$, los datos se cargan en los flip-flops desde el bus. En vez de utilizar multiplexores en las entradas de los flip-flops,

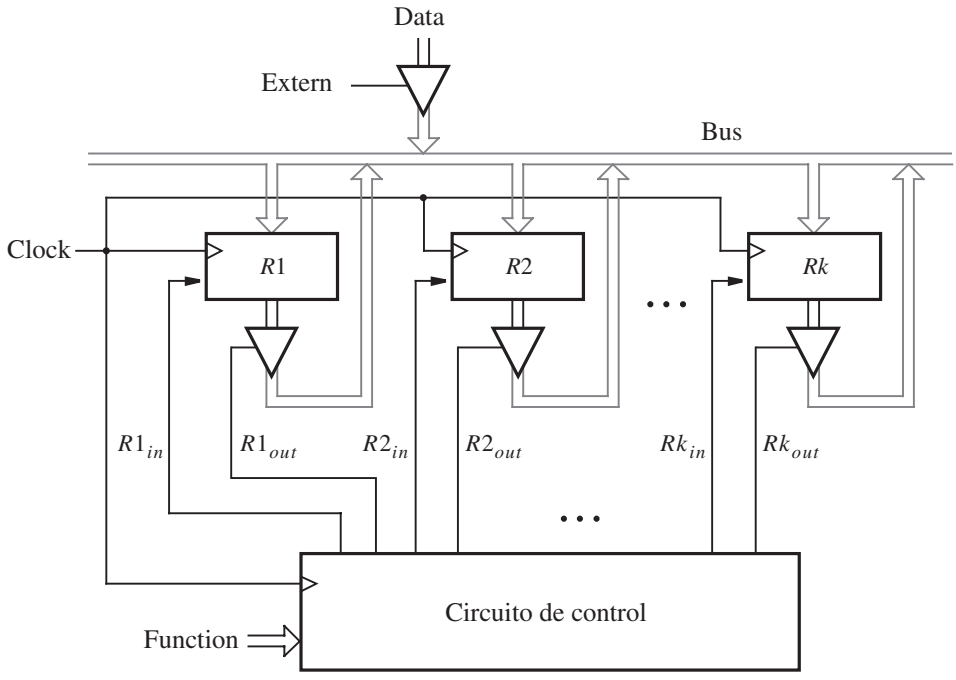


Figura 7.55 Un sistema digital con registros *k*.

podría intentarse conectar las entradas D en los flip-flops directamente al bus. Entonces es necesario controlar las entradas del reloj en todos los flip-flops para asegurar que están sincronizadas sólo cuando deben cargarse datos nuevos en el registro. Este enfoque no es conveniente porque puede ocurrir que diferentes flip-flops se sincronicen en tiempos ligeramente distintos, lo que causaría un problema conocido como *desviación del reloj (clock skew)*. Un análisis detallado de los problemas relacionados con la sincronización de los flip-flops se proporciona en la sección 10.3.

El sistema de la figura 7.55 puede utilizarse de muchas formas, según el diseño del circuito de control y de cuántos registros y otros bloques de circuitos estén conectados al bus. Como ejemplo simple considérese un sistema con tres registros, $R1$, $R2$ y $R3$. Cada uno de ellos está conectado al bus como se indicó en la figura 7.56. Diseñaremos un circuito de control que cumple una sola función: intercambia el contenido de los registros $R1$ y $R2$, y usa $R3$ para almacenamiento temporal.

El intercambio requerido se realiza en tres pasos, cada uno de los cuales necesita un ciclo del reloj. En el primer paso el contenido de $R2$ se transfiere a $R3$. Luego el contenido de $R1$ se transfiere a $R2$. Finalmente, el contenido de $R3$, que tiene el contenido original de $R2$, se transfiere a $R1$. Nótese que decimos que el contenido de un registro, R_i, se "transfiere" a otro registro, R_j. Esta terminología suele usarse para indicar que el nuevo contenido de R_j será una copia del contenido de R_i. El contenido de R_i no cambia como resultado de la transferencia. Por consiguiente, sería más preciso decir que el contenido de R_i se "copia" en R_j.

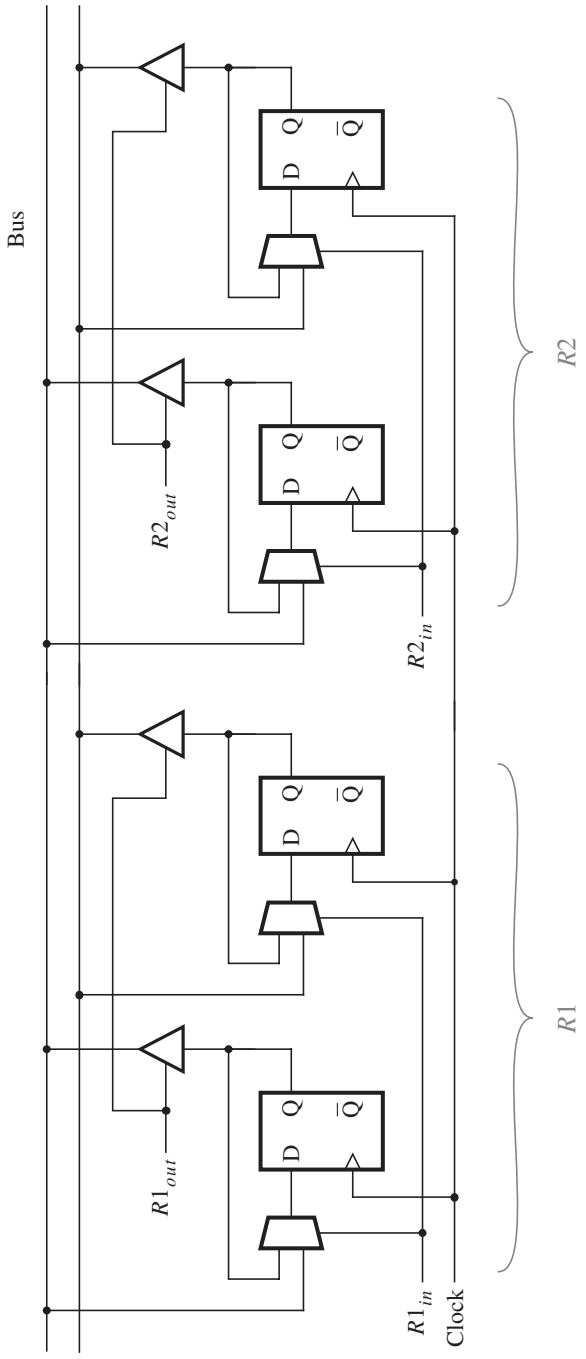


Figura 7.56 Detalles para conectar los registros a un bus.

Uso de un registro de corrimiento para control

Hay muchas formas de diseñar un circuito de control adecuado para la operación de intercambio. Una posibilidad es usar el registro de corrimiento de izquierda a derecha mostrado en la figura 7.57. Supóngase que la entrada reset se emplea para borrar los flip-flops y establecerlos en 0. Por tanto, las señales de control $R1_{in}$, Rl_{out} y demás no se validan, ya que las salidas del registro de corrimiento tienen el valor 0. La entrada serial w normalmente tiene el valor 0. Suponemos que los cambios en el valor de w están sincronizados para que ocurran poco tiempo después del flanco activo del reloj. Tal suposición es razonable porque comúnmente w se genera como la salida de algún circuito controlado por la misma señal de reloj. Cuando el intercambio buscado debe realizarse, w se establece en 1 para un ciclo del reloj, y luego regresa a 0. Después del siguiente flanco activo del reloj, la salida del flip-flop en el extremo izquierdo se vuelve igual a 1, lo cual valida tanto $R2_{out}$ como $R3_{in}$. El contenido del registro $R2$ se coloca en los cables del bus y se carga en un registro $R3$ en el siguiente flanco activo del reloj. Este flanco del reloj también desplaza el contenido del registro de corrimiento, con lo cual $R1_{out} = R2_{in} = 1$. Obsérvese que como w ahora es 0, el primer flip-flop se borra, lo que ocasiona que $R2_{out} = R3_{in} = 0$. El contenido de $R1$ ahora está en el bus y se carga en $R2$ en el siguiente flanco del reloj. Después de este flanco, el registro de corrimiento contiene 001 y, por ende, valida $R3_{out}$ y $R1_{in}$. El contenido de $R3$ ahora está en el bus y se carga en $R1$ en el siguiente flanco del reloj.

Al usar el circuito de control de la figura 7.57, cuando w cambia a 1 la operación de intercambio no comienza sino hasta después del siguiente flanco activo del reloj. Podemos modificar el circuito de control de modo que la operación de intercambio comience en el mismo ciclo del reloj en el que w cambia a 1. Un enfoque posible se ilustra en la figura 7.58. La señal reset se utiliza para establecer en 100 el contenido del registro de corrimiento, al prestablecer el flip-flop del extremo izquierdo en 1 y borrar los otros dos flip-flops. Siempre que $w = 0$, las señales de control de salida no se validan. Cuando w cambia a 1, las señales $R2_{out}$ y $R3_{in}$ se validan de inmediato y el contenido de $R2$ se coloca en el bus. El siguiente flanco activo del reloj carga estos datos en $R3$ y también desplaza el contenido del registro de corrimiento a 010. Puesto que la señal $R1_{out}$ ahora se valida, el contenido de $R1$ aparece en el bus. El siguiente flanco del reloj carga estos datos en $R2$ y cambia el contenido del registro de corrimiento a 001. El contenido de $R3$ ahora está en el bus; estos datos se cargan en $R1$ en el siguiente flanco del reloj, el cual también cambia el contenido del registro de corrimiento a 100. Suponemos que w tuvo el valor de 1 sólo para un ciclo del reloj; por consiguiente, las señales de control de la salida no se validan en este

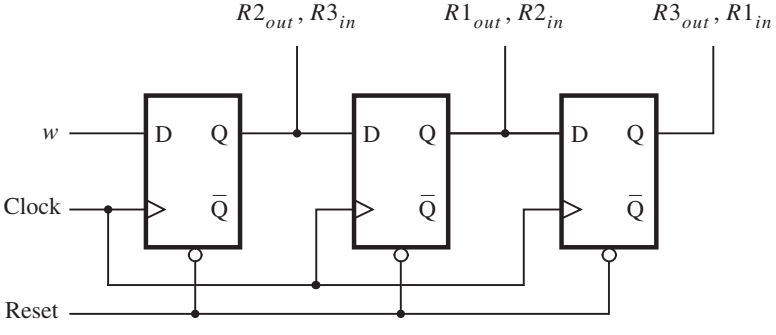


Figura 7.57 Un circuito de control del registro de corrimiento.

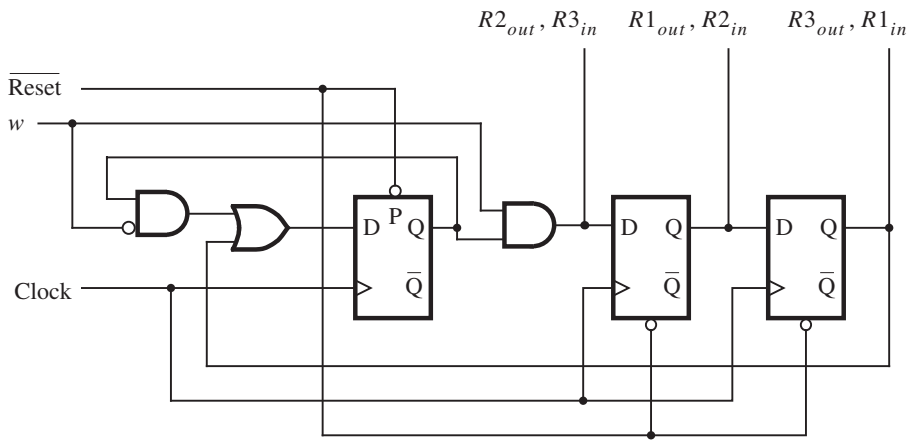


Figura 7.58 Un circuito de control modificado.

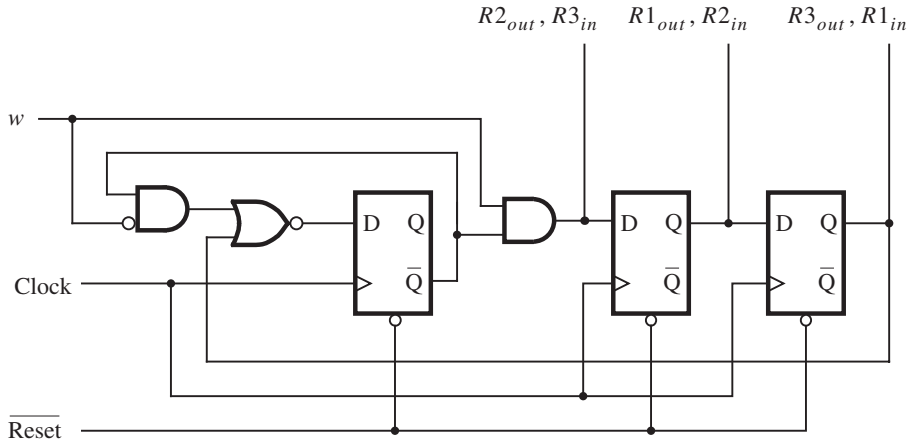


Figura 7.59 Una versión modificada del circuito de la figura 7.58.

punto. Tal vez no sea evidente para el lector cómo diseñar un circuito como el mostrado en la figura 7.58 debido a que hemos presentado el diseño según lo requiere el caso. En la sección 8.3 mostraremos cómo diseñar este circuito utilizando un enfoque más formal.

El circuito de la figura 7.58 supone que una entrada preset está disponible en el flip-flop del extremo izquierdo. Si el flip-flop tiene sólo una entrada clear, entonces podemos usar el circuito equivalente que aparece en la figura 7.59. En este circuito usamos la salida $\overline{Q}$ del flip-flop del extremo izquierdo y también complementamos la entrada a este flip-flop con una compuerta NOR en vez de una OR.

Uso de un multiplexor para implementar un bus

En la figura 7.55 usamos buffers triestado para controlar el acceso al bus. Un enfoque opcional consiste en utilizar multiplexores, como se describe en la figura 7.60. Las salidas de cada registro están conectadas a un multiplexor. La salida de este multiplexor está conectada a las entradas de los registros, con lo que se crea el bus. Las entradas select del multiplexor determinan el contenido de cuál registro aparece en el bus. Aun cuando la figura muestra sólo un símbolo de multiplexor, en realidad necesitamos un multiplexor por cada bit de los registros, $R1$ a $R4$, además de la entrada *Data* de ocho bits suministrada externamente. Para interconectarlas requerimos ocho multiplexores cinco a uno. En la figura 7.57 usamos un registro de corrimiento para implementar el circuito de control. Un enfoque similar puede utilizarse con los multiplexores. Las señales que controlan cuándo se cargan los datos en un registro, como $R1_{in}$, aún pueden conectarse directamente a las salidas del registro de corrimiento. Sin embargo, en vez de usar señales de control como $R1_{out}$ para colocar el contenido de un registro en el bus, hemos de generar las entradas select para los multiplexores. Una forma de hacerlo es conectar las salidas del registro de corrimiento a un circuito codificador que produce las entradas select para el multiplexor. Estudiamos los circuitos codificadores en la sección 6.3.

Los enfoques de buffer triestado y de multiplexor para implementar el bus son igualmente válidos. No obstante, ciertos tipos de chips, como el grueso de los PLD, no contienen un número suficiente de buffers triestado para producir buses incluso de tamaño mediano. En estos chips el enfoque del multiplexor es la única opción real. En la práctica, los circuitos se diseñan con herramientas CAD. Si el diseñador describe el circuito mediante buffers triestado pero no hay suficientes buffers en el dispositivo objetivo, entonces las herramientas CAD producen de manera automática un circuito equivalente que utiliza multiplexores.

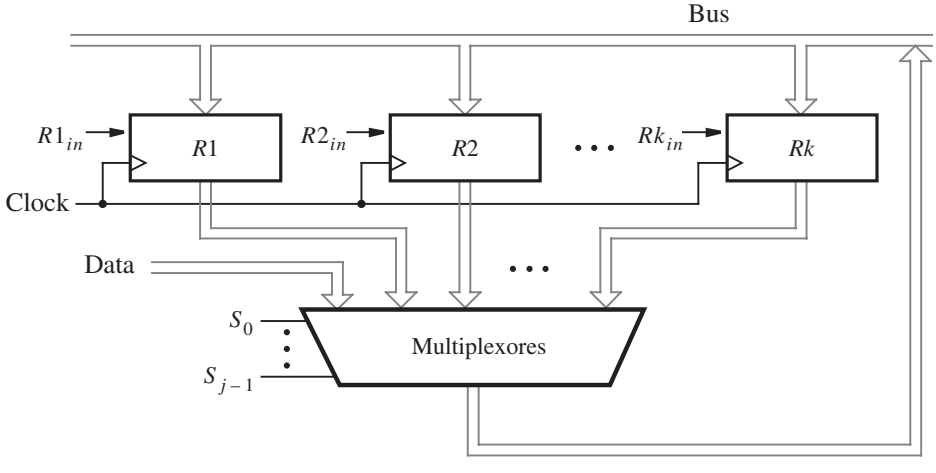


Figura 7.60 Uso de multiplexores para implementar un bus.

Código de VHDL

En esta sección se presenta el código de VHDL para nuestro circuito de ejemplo que intercambia el contenido de dos registros. Primero damos el código para el estilo de circuito de la figura 7.55 que utiliza buffers triestado para implementar el bus y luego ofrecemos el código para el estilo de circuito de la figura 7.60, que emplea multiplexores. El código está escrito de una manera jerárquica, usando subcircuitos para los registros, buffers triestado y el registro de corrimiento. En la figura 7.61 se presenta el código para un registro de *n* bits del tipo de la figura 7.56. El número de bits del registro se establece por medio del parámetro genérico *N*, el cual tiene el valor predeterminado de 8. El proceso que describe el registro especifica que la entrada *Rin* = 1, entonces los flip-flops se cargan desde la entrada *R* de *n* bits. De lo contrario, los flip-flops conservan sus valores almacenados en ese momento. El circuito sintetizado a partir de este código tiene un multiplexor dos a uno controlado por *Rin* conectado a la entrada *D* en cada flip-flop, como se describe en la figura 7.56.

En la figura 7.62 aparece el código para un subcircuito que representa *n* buffers triestado, cada uno habilitado por la entrada *E*. El número de buffers se establece mediante el parámetro genérico *N*. Las entradas a los buffers son la señal *X* de *n* bits y las salidas son la señal *F* de *n* bits. La arquitectura utiliza la sintaxis (OTHERS => 'Z') para especificar que la salida de cada buffer establece el valor de Z si *E* = 0; de lo contrario, la salida se establece como *F* = *X*.

En la figura 7.63 se proporciona el código de un registro de corrimiento que puede utilizarse para implementar el circuito de control de la figura 7.57. El número de flip-flops se establece por medio del parámetro genérico *K*, el cual tiene el valor predeterminado de 4. El registro de corrimiento tiene una entrada reset asíncrona activa en nivel bajo. La operación de corrimiento se define con un FOR LOOP en el estilo usado en el ejemplo 7.9.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( R          : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Rin, Clock : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Rin = '1' THEN
            Q <= R ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.61 Código para un registro de *n* bits del tipo de la figura 7.56.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY trin IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           E  : IN   STD_LOGIC ;
           F  : OUT  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END trin ;

ARCHITECTURE Behavior OF trin IS
BEGIN
    F <= (OTHERS => 'Z') WHEN E = '0' ELSE X ;
END Behavior ;
```

Figura 7.62 Código para un buffer triestado de *n* bits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shiftr IS - - registro de corrimiento de izquierda a derecha con reset asíncrono
    GENERIC ( K : INTEGER := 4 ) ;
    PORT ( Resetn, Clock, w  : IN          STD_LOGIC ;
           Q                 : BUFFER  STD_LOGIC_VECTOR(1 TO K) ) ;
END shiftr ;

ARCHITECTURE Behavior OF shiftr IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Genbits: FOR i IN K DOWNTO 2 LOOP
                Q(i) <= Q(i−1) ;
            END LOOP ;
            Q(1) <= w ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 7.63 Código para el registro de corrimiento de la figura 7.57.

Para utilizar como subcircuitos las entidades de las figuras 7.61 a 7.63 debemos proporcionar dos declaraciones de componentes por cada una. Por comodidad, las colocamos dentro de un solo paquete llamado *components*, el cual se muestra en la figura 7.64. Este paquete se utiliza en el código de la figura 7.65. Representa el sistema digital de la figura 7.55 con tres registros de ocho bits, *R*1, *R*2 y *R*3.

El circuito de la figura 7.55 incluye buffers triestado que se ocupan para colocar *n* bits de datos suministrados externamente en el bus. En el código de la figura 7.65, estos buffers se instancian en la instrucción etiquetada *tri_ext*. Cada uno de los ocho buffers se habilita mediante la señal de entrada *Extern*, y las entradas de datos en los buffers se conectan a la señal *Data* de ocho bits. Cuando *Extern* = 1, el valor de *Data* se coloca en el bus, que está representado por la señal *BusWires*. El puerto *BusWires* representa la salida del circuito. Este puerto tiene el modo INOUT, que es necesario porque *BusWires* está conectado a las salidas de los buffers triestado, los cuales a su vez están conectados a las entradas de los registros.

Suponemos que existe una señal de control de tres bits llamada *RinExt*, la cual se utiliza para permitir que los datos suministrados externamente se carguen desde el bus en los registros *R*1,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    COMPONENT regn - - registro
        GENERIC ( N : INTEGER := 8 ) ;
        PORT ( R            : IN    STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
               Rin, Clock   : IN    STD_LOGIC ;
               Q            : OUT   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT shiftr - - registro de corrimiento de izquierda a derecha con reset asíncrono
        GENERIC ( K : INTEGER := 4 ) ;
        PORT ( Resetn, Clock, w  : IN          STD_LOGIC ;
               Q                 : BUFFER  STD_LOGIC_VECTOR(1 TO K) ) ;
    END component ;

    COMPONENT trin - - buffers triestado
        GENERIC ( N : INTEGER := 8 ) ;
        PORT ( X   : IN    STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
               E   : IN    STD_LOGIC ;
               F   : OUT   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
    END COMPONENT ;

END components ;
```

Figura 7.64 Paquete y declaraciones de componentes.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY swap IS
    PORT ( Data          : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, w     : IN    STD_LOGIC ;
           Clock, Extern : IN    STD_LOGIC ;
           RinExt        : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           BusWires      : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swap ;

ARCHITECTURE Behavior OF swap IS
    SIGNAL Rin, Rout, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
    control: shiftr GENERIC MAP ( K => 3 )
                    PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;
    Rout(1) <= Q(2) ; Rout(2) <= Q(1) ; Rout(3) <= Q(3) ;

    tri_ext: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
END Behavior ;
```

Figura 7.65 Un sistema digital como el de la figura 7.55.

*R*2 o *R*3. La entrada *RinExt* no se muestra en la figura 7.55 para mantener la simplicidad de ésta, pero se generaría por el mismo bloque de circuito externo que produce *Extern* y *Data*. Cuando *RinExt*(1) = 1, los datos en el bus se cargan en el registro *R*1; cuando *RinExt*(2) = 1, los datos se cargan en *R*2 y cuando *RinExt*(3) = 1, los datos se cargan en *R*3.

En la figura 7.65 el registro de corrimiento de tres bits se instancia en la instrucción etiquetada *control*. Las salidas del registro de corrimiento son la señal Q de tres bits. Las tres instrucciones siguientes conectan Q a las señales de control que determinan cuándo se cargan los datos en cada registro, los cuales están representados por la señal *Rin* de tres bits. Las señales *Rin*(l), *Rin*(2) y *Rin*(3) en el código corresponden a las señales $R1_{in}$, $R2_{in}$ y $R3_{in}$ de la figura 7.55. Como se especificó en la figura 7.57, la salida del registro de corrimiento del extremo izquierdo, Q(1), controla

cuándo se cargan los datos en el registro *R*3. De modo similar, Q(2) controla el registro *R*2 y Q(3) controla *R*1. A cada bit en *Rin* se le suma (OR) correspondiente en *RinExt*, de modo que los datos suministrados externamente puedan almacenarse en los registros, como se vio antes. El código también conecta las salidas del registro de corrimiento para habilitar las entradas, llamadas *Rout*, en los buffers triestado que conectan los registros con el bus. En la figura 7.57 se muestra que Q(1) se usa para poner el contenido de *R*2 en el bus; por consiguiente, el valor de Q(l) se asigna a *Rout*(2). De manera similar, el valor de Q(2) se asigna a *Rout*(1) y el valor de Q(3), a *Rout*(3). Las instrucciones restantes del código instancian los registros y los buffers triestado del sistema.

Código de VHDL que utiliza multiplexores

En la figura 7.66 se indica cómo el código de la figura 7.65 puede modificarse para usar multiplexores en vez de buffers triestado. Al usar la estructura del circuito mostrada en la figura 7.60, el bus se implementa utilizando ocho multiplexores cuatro a uno. Tres de las entradas de datos de cada multiplexor cuatro a uno se conectan a un bit desde los registros *R*1, *R*2 y *R*3. La cuarta entrada de datos se conecta a un bit de la señal de entrada *Data* para permitir que los datos suministrados externamente se escriban en los registros. Cuando el contenido del registro de corrimiento es 000, los multiplexores seleccionan *Data* para colocarla en el bus. Estos datos se cargan en el registro seleccionado por *RinExt*. Se cargan en *R*1 si *RinExt*(1) = 1, en *R*2 si *RinExt*(2) = 1 y en *R*3 si *RinExt*(3) = 1.

La señal *Rout* de la figura 7.65, la cual se utiliza como la entrada enable del buffer triestado conectado al bus, no se necesita para la implementación del multiplexor. En vez de ello, deben proporcionarse las entradas select en los multiplexores. En el cuerpo de arquitectura de la figura 7.66 las salidas del registro de corrimiento se llaman Q. Estas señales se utilizan para generar las señales de control *Rin* para los registros de la misma manera que se muestra en la figura 7.65. En el análisis referente a la figura 7.60 dijimos que se necesita un codificador entre las salidas del registro de corrimiento y las salidas select del multiplexor. Un codificador adecuado se describe en la asignación de la señal seleccionada etiquetada *encoder*. Produce las entradas select del multiplexor, las cuales se llaman *S*. Establece *S* = 00 cuando el registro de corrimiento contiene 000, *S* = 10 cuando el registro de corrimiento contiene 100 y así por el estilo, como se muestra en el código. Los multiplexores se describen mediante la asignación de la señal seleccionada etiquetada *muxes*. Esta instrucción coloca el valor de *Data* en el bus (*BusWires*) si *S* = 00, el contenido del registro *R*1 si *S* = 01, y así sucesivamente. Al usar este esquema, cuando la operación de intercambio no está activa, los multiplexores colocan los bits desde la entrada *Data* en el bus.

En la figura 7.66 empleamos dos asignaciones de señal seleccionada, una para describir un codificador y la otra para describir los multiplexores de bus. Un enfoque más sencillo consiste en utilizar una sola asignación de señal, como se muestra en la figura 7.67. La instrucción etiquetada *muxes* especifica cuál señal debe aparecer en *BusWires* para cada patrón de las salidas del registro de corrimiento. El circuito sintetizado a partir de esta instrucción es similar a un multiplexor ocho a uno con las tres entradas select conectadas a las salidas del registro de corrimiento. No obstante, en realidad sólo la mitad del circuito multiplexor se genera por medio de las herramientas de síntesis debido a que sólo hay cuatro entradas de datos. El circuito producido a partir del código de la figura 7.67 es el mismo que el generado por el de la figura 7.66.

En la figura 7.68 se muestra un ejemplo de una simulación de tiempo para un circuito sintetizado con base en el código de la figura 7.67. En la primera mitad de la simulación el circuito se inicializa, lo mismo que el contenido de los registros *R*1 y *R*2. El valor hexadecimal 55 se carga en *R*1 y el valor AA se carga en *R*2. El flanco del reloj en 275 ns, marcado por la línea de referencia vertical de la figura 7.68, carga el valor de *w* = 1 en el registro de corrimiento. El

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;
ENTITY swapmux IS
    PORT ( Data      : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, w : IN     STD_LOGIC ;
           Clock     : IN     STD_LOGIC ;
           RinExt    : IN     STD_LOGIC_VECTOR(1 TO 3) ;
           BusWires  : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swapmux ;

ARCHITECTURE Behavior OF swapmux IS
    SIGNAL Rin, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL S : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
    control: shiftr GENERIC MAP ( K => 3 )
                    PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;

    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    encoder:
    WITH Q SELECT
        S <= "00" WHEN "000",
             "10" WHEN "100",
             "01" WHEN "010",
             "11" WHEN OTHERS;
    muxes: --ocho multiplexores cuatro a uno
    WITH S SELECT
        BusWires <= Data WHEN "00",
                    R1 WHEN "01",
                    R2 WHEN "10",
                    R3 WHEN OTHERS ;
END Behavior ;
```

Figura 7.66 Uso de multiplexores para implementar un bus.

```
ARCHITECTURE Behavior OF swapmux IS
    SIGNAL Rin, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
    control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;

    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;

    muxes:
    WITH Q SELECT
        BusWires <= Data WHEN "000",
                    R2 WHEN "100",
                    R1 WHEN "010",
                    R3 WHEN OTHERS ;
END Behavior ;
```

Figura 7.67 Una versión simplificada de la arquitectura de la figura 7.66.

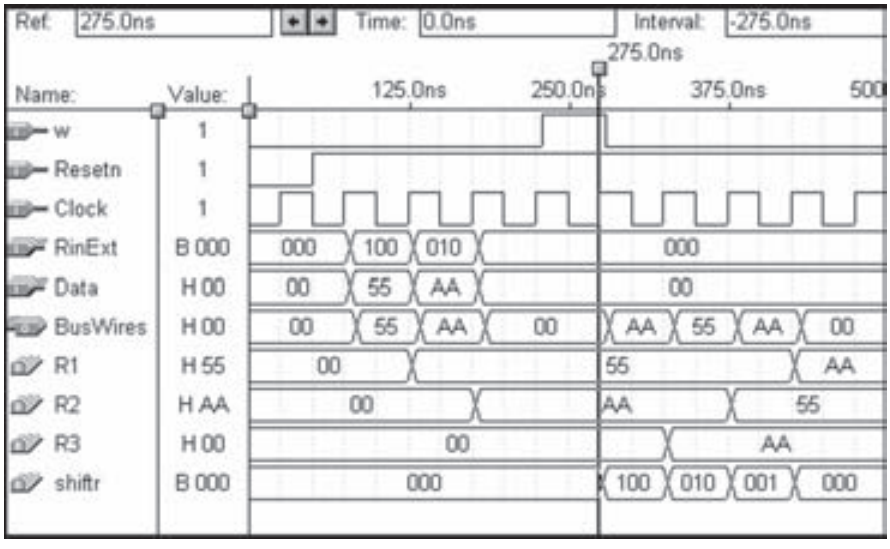


Figura 7.68 Simulación de tiempo para el código de VHDL de la figura 7.67.

contenido de *R*2 (AA) aparece entonces en el bus y se carga en *R*3 cuando el flanco del reloj está en 325 ns. Después de este flanco, el contenido del registro de corrimiento es 010 y los datos almacenados en *R*1 (55) están en el bus. El flanco del reloj en 375 ns carga estos datos en *R*2 y cambia el registro de corrimiento a 001. El contenido de *R*3(AA) ahora aparece en el bus y se carga en *R*1 cuando el flanco del reloj está en 425 ns. El registro de corrimiento está ahora en el estado 000 y el intercambio está completo.

7.14.2 PROCESADOR SIMPLE

Un segundo ejemplo de un sistema digital como el de la figura 7.55 se muestra en la figura 7.69. Tiene cuatro registros de *n* bits, *R*0,..., *R*3, que están conectados al bus mediante buffers triestado. Los datos externos pueden cargarse en los registros desde la entrada *Data* de *n* bits, la cual se conecta al bus por buffers triestado habilitados por medio de la señal de control *Extern*. El sistema también incluye un módulo sumador/restador. Una de sus entradas de datos es provista por un registro de *n* bits, *A*, que está conectado al bus, mientras que la otra entrada de datos, *B*, está directamente conectada al bus. Si la señal *AddSub* tiene el valor 0, el módulo genera la suma $A + B$; si $AddSub = 1$, el módulo genera la diferencia $A - B$. Para realizar la sustracción, suponemos que el sumador/restador incluye las compuertas XOR requeridas para formar el complemento a 2 de *B*, como se vio en la sección 5.3. El registro *G* almacena la salida producida por el sumador/restador. Los registros *A* y *G* están controlados por las señales A_{in}, G_{in} y G_{out}.

El sistema de la figura 7.69 puede realizar varias funciones, según el diseño del circuito de control. Como ejemplo, diseñaremos un circuito de control que pueda cumplir las cuatro acciones indicadas en la tabla 7.2. La columna izquierda muestra el nombre de cada operación y sus operandos; la derecha indica la función realizada en la operación. Para la operación *Load* el significado de $Rx \leftarrow Data$ es que los datos de la entrada *Data* externa se transfieren a través del bus a cualquier registro, *Rx*, donde *Rx* puede ser de *R*0 a *R*3. La operación *Move* copia los datos almacenados en el registro *Ry* en el registro *Rx*. En la tabla, los corchetes, como en [*Rx*], se refieren al *contenido* de un registro. Puesto que sólo se precisa una transferencia a través del bus, las operaciones *Load* y *Move* requieren sólo un paso (ciclo del reloj) para completarse. Las operaciones *Add* y *Sub* necesitan tres pasos, como sigue: en el primero el contenido de *Rx* se transfiere a través del bus en un registro *A*. Luego, en el paso siguiente el contenido de *Ry* se coloca en el bus. El módulo sumador/restador realiza la función requerida y los resultados se guardan en el registro *G*. Por último, en el tercer paso el contenido de *G* se transfiere a *Rx*.

Un sistema digital que realiza los tipos de operaciones señalados en la tabla 7.2 se conoce como *procesador*. La operación específica que va a llevarse a cabo en un momento dado se indica mediante la entrada del circuito de control llamada *Function*. La operación se inicia al establecer *w* en 1, y el circuito de control valida la salida *Done* cuando la operación se completa.

En la figura 7.55 usamos un registro de corrimiento para implementar el circuito de control. Es posible emplear un diseño similar para el sistema de la figura 7.69. A fin de ilustrar un enfoque distinto basaremos el diseño del circuito de control en un contador. Este circuito debe generar las señales de control requeridas en cada paso de cada operación. Como las operaciones más largas (*Add* y *Sub*) necesitan tres pasos (ciclos del reloj), puede usarse un contador de dos bits. En la figura 7.70 se muestra un contador de dos bits conectado a un decodificador dos a cuatro. Los decodificadores se estudiaron en la sección 6.2. El decodificador se habilita en todo momento

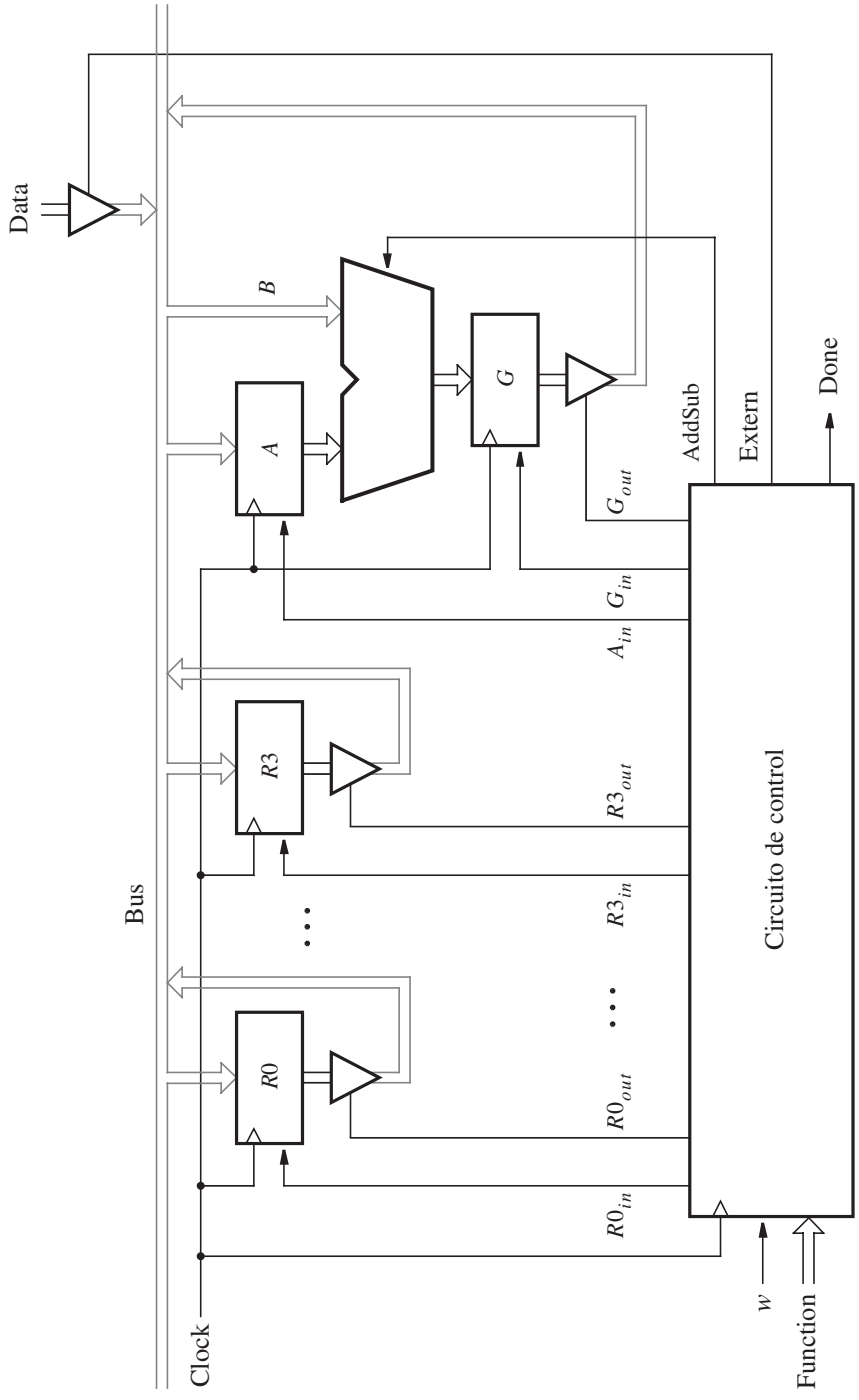


Figura 7.69 Sistema digital que implementa un procesador simple.

Tabla 7.2 Operaciones realizadas en el procesador.

Operación	Función realizada
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

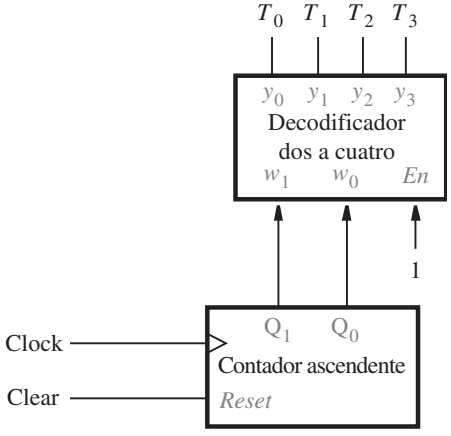


Figura 7.70 Parte de un circuito de control para el procesador.

al establecer de modo permanente su entrada de habilitación (*En*) en el valor 1. Cada una de las salidas del decodificador representa un paso de una operación. Cuando ninguna operación se está realizando en un momento dado, el valor de conteo es 00; por tanto, la salida T_0 del decodificador se valida. En el primer paso de una operación, el valor de conteo es 01 y T_1 se valida. Durante el segundo y tercer pasos de las operaciones *Add* y *Sub*, T_2 y T_3 se validan, respectivamente.

En cada uno de los pasos T_0 a T_3, varios valores de la señal de control deben generarse por medio del circuito de control, según la operación que vaya a realizarse. En la figura 7.71 se muestra que la operación se especifica con seis bits, los cuales forman la entrada *Function*. Los dos bits del extremo izquierdo, $F = f_1 f_0$ se usan como un número de dos bits que identifica la operación. Para representar *Load*, *Move*, *Add* y *Sub* usamos los códigos $f_1 f_0 = 00, 01, 10$ y 11, respectivamente. Las entradas $Rx_1 Rx_0$ son un número binario que identifica al operando *Rx*, mientras que $Ry_1 Ry_0$ identifica al operando *Ry*. Las entradas *Function* se almacenan en un registro de funciones de seis bits cuando la señal FR_{in} se valida.

En la figura 7.71 también se muestran tres decodificadores dos a cuatro que sirven para decodificar la información codificada en las entradas *F*, *Rx* y *Ry*. En breve veremos que estos

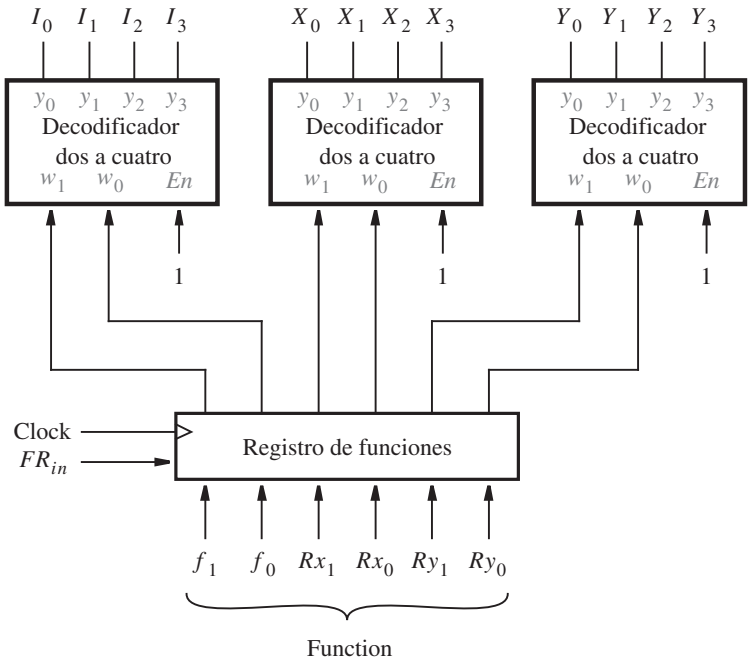


Figura 7.71 El registro de funciones y los decodificadores.

decodificadores se incluyen por conveniencia debido a que sus salidas proporcionan expresiones lógicas de apariencia sencilla para las diversas señales de control.

Los circuitos de las figuras 7.70 y 7.71 forman una parte del circuito de control. Mostraremos cómo derivar el resto del circuito de control usando la entrada w y las señales T_0,..., T_3, I_0,..., I_3, X_0,..., X_3 y Y_0,..., Y_3. Debe generar las salidas *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}$,..., $R3_{in}$ y $R0_{out}$,..., $R3_{out}$. El circuito de control también debe generar las señales *Clear* y FR_{in} utilizadas en las figuras 7.70 y 7.71.

Clear y FR_{in} están definidas de la misma manera para todas las operaciones. *Clear* se usa para asegurar que el valor de conteo permanezca en 00 siempre que $w = 0$ y ninguna operación se esté ejecutando. También sirve para reiniciar el valor de conteo a 00 al final de cada operación. Por tanto, una expresión lógica apropiada es

$$Clear = \overline{w}\, T_0 + Done$$

La señal FR_{in} se utiliza para cargar los valores de las entradas *Function* en el registro de funciones cuando w cambia a 1. Por consiguiente,

$$FR_{in} = wT_0$$

El resto de las salidas del circuito de control depende del paso específico que vaya a efectuarse en cada operación. Los valores que deben generarse para cada señal se muestran en la tabla 7.3. Cada fila de la tabla corresponde a una operación específica, y cada columna representa un paso

Tabla 7.3 Señales de control validadas en cada operación/paso de tiempo

	T_1	T_2	T_3
(Load): I_0	*Extern*, $R_{in} = X$, *Done*		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, *Done*		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in} *AddSub* = 0	G_{out}, $R_{in} = X$, *Done*
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in} *AddSub* = 1	G_{out}, $R_{in} = X$, *Done*

de tiempo. La señal *Extern* se valida sólo en el primer paso de la operación *Load*. Por consiguiente, la expresión lógica que implementa esta señal es

$$Extern = I_0 T_1$$

Done se valida en el primer paso de *Load* y *Move*, así como en el tercer paso de *Add* y *Sub*. Por tanto

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

Las señales A_{in}, G_{in} y G_{out} se validan en las operaciones *Add* y *Sub*. A_{in} se valida en el paso T_1; G_{in} en T_2, y G_{out} en T_3. La señal *AddSub* debe establecerse en 0 en la operación *Add* y en 1 en la operación *Sub*. Ello se logra con las expresiones lógicas siguientes

$$A_{in} = (I_2 + I_3)T_1$$
$$G_{in} = (I_2 + I_3)T_2$$
$$G_{out} = (I_2 + I_3)T_3$$
$$AddSub = I_3$$

Los valores de $R0_{in}$,..., $R3_{in}$ se determinan utilizando ya sea las señales X_0,..., X_3 o las señales Y_0,..., Y_3. En la tabla 7.3 estas acciones se indican escribiendo $R_{in} = X$ o $R_{in} = Y$. El significado de $R_{in} = X$ es que $R0_{in} = X_0$, $R1_{in} = X_1$ y así sucesivamente. De modo similar, los valores de $R0_{out}$,..., $R3_{out}$ se especifican utilizando ya sea $R_{out} = X$ o $R_{out} = Y$.

Desarrollaremos las expresiones para $R0_{in}$ y $R0_{out}$ al examinar la tabla 7.3 y luego mostraremos cómo derivar las expresiones para las otras señales de control del registro. En la tabla se muestra que $R0_{in}$ se establece en el valor de X_0 en el primer paso de las operaciones *Load* y *Move*, y en el tercer paso de las operaciones *Add* y *Sub*, lo cual nos lleva a la expresión

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

De forma similar, $R0_{out}$ se establece en el valor de Y_0 en el primer paso de *Move*. Se establece en X_0 en el primer paso de *Add* y *Sub*, y en Y_0 en el segundo paso de estas operaciones, lo que da

$$R0_{out} = I_1 T_1 Y_0 + (I_2 + I_3)(T_1 X_0 + T_2 Y_0)$$

Las expresiones para $R1_{in}$ y $R1_{out}$ son las mismas que aquellas para $R0_{in}$ y $R0_{out}$, excepto que X_1 y Y_1 se usan en lugar de X_0 y Y_0. Las expresiones para $R2_{in}$, $R2_{out}$, $R3_{in}$ y $R3_{out}$ se derivan de la misma forma.

Los circuitos mostrados en las figuras 7.70 y 7.71, combinados con los circuitos representados por las expresiones anteriores, implementan el circuito de control de la figura 7.69.

Los procesadores son circuitos sumamente útiles de uso muy común. Hemos presentado sólo los aspectos más básicos de su diseño. Sin embargo, las técnicas expuestas pueden ampliarse para diseñar procesadores reales, como los microprocesadores modernos. El lector interesado puede referirse a los libros referentes a la organización de computadoras para que conozca más detalles del diseño de procesadores [1-2].

Código de VHDL

En esta sección damos dos estilos diferentes de código de VHDL para describir el sistema de la figura 7.69. El primero utiliza buffers triestado para representar el bus y da las expresiones lógicas mostradas líneas arriba para las salidas del circuito de control. El segundo estilo de código emplea multiplexores para representar el bus y usa instrucciones CASE que corresponden a la tabla 7.3 a fin de describir las salidas del circuito de control.

El código de VHDL para un contador ascendente se muestra en la figura 7.52. Una versión modificada de este contador, llamada *upcount*, se muestra en el código de la figura 7.72. Tiene una entrada reset síncrona, que está activa en nivel alto. En la figura 7.64 definimos el paquete llamado *components*, el cual proporciona declaraciones de componentes para varios subcircuitos. En el código de VHDL para el procesador usaremos los componentes *regn* y *trin* enumerados en la figura 7.64, pero no el componente *shiftr*. Creamos un paquete nuevo llamado *subccts* para usarlo con el procesador. El código no se muestra aquí, pero incluye declaraciones de componentes para *regn* (figura 7.61), *trin* (figura 7.62), *upcount* y *dec2to4* (figura 6.30).

El código completo para el procesador se presenta en la figura 7.73. En el cuerpo de arquitectura, las instrucciones etiquetadas *counter* y *decT* instancian los subcircuitos de la figura 7.70. Obsérvese que hemos supuesto que el circuito tiene una entrada de inicialización activa en nivel alto, *Reset*, que se utiliza para inicializar el contador en 00. La instrucción Func <= F & Rx & Ry usa el operador de concatenación para crear la señal de seis bits *Func*, la cual representa las entradas al registro de funciones de la figura 7.71. La siguiente instrucción instancia el registro de funciones con las entradas de datos *Func* y las salidas *FuncReg*. Las instrucciones etiquetadas *decI*, *decX* y *decY* instancian los decodificadores de la figura 7.71. Después de estas instrucciones se dan las expresiones lógicas derivadas antes para las salidas del circuito de control. Para $R0_{in}$, . . . , $R3_{in}$ y $R0_{out}$, . . . , $R3_{out}$, una instrucción GENERATE se usa para producir las expresiones.

Al final del código, los buffers triestado y los registros del procesador se instancian, y el módulo sumador/restador se describe con una asignación de la señal seleccionada.

Uso de multiplexores e instrucciones CASE

En la figura 7.60 mostramos que un bus puede implementarse usando multiplexores en vez de buffers triestado. El código de VHDL que describe al procesador usando este enfoque

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clear, Clock  : IN      STD_LOGIC ;
           Q             : BUFFER  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    upcount: PROCESS ( Clock )
    BEGIN
        IF (Clock'EVENT AND Clock = '1') THEN
            IF Clear = '1' THEN
                Q <= "00" ;
            ELSE
                Q <= Q + '1' ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figura 7.72 Código para un contador ascendente de dos bits con reset síncrono.

aparece en la figura 7.74. La misma declaración de entidad dada en la figura 7.73 puede usarse y no se muestra en la figura 7.74. El código ilustra una manera diferente de describir el circuito de control en el procesador. No da expresiones lógicas para las señales *Extern*, *Done* y el resto, como sí ocurre en la figura 7.73. En vez de ello, se emplean instrucciones CASE para representar la información mostrada en la tabla 7.3. Estas instrucciones se proporcionan dentro del proceso llamado *controlsignals*. A cada señal de control se asigna primero el valor de 0, en forma predeterminada. Esto es necesario porque las instrucciones CASE especifican los valores de las señales de control sólo cuando deben validarse, como se hizo en la tabla 7.3. Según se explicó para la figura 7.35, cuando el valor de una señal no está especificado, la señal conserva su valor presente. Esta memoria implícita da como resultado una conexión de retroalimentación en el circuito sintetizado. Evitamos este problema al proporcionar el valor predeterminado de 0 para cada una de las señales de control implícitas en las instrucciones CASE.

En la figura 7.73 las instrucciones etiquetadas *decT* y *decI* se usan para decodificar la señal *Count* y los valores almacenados de la entrada *F*, respectivamente. El decodificador *decT* tiene las salidas $T_0, \ldots, T_3$ y *decI* produce $I_0, \ldots, I_3$. En la figura 7.74 estos dos decodificadores no se utilizan, ya que no sirven para un propósito útil en este código. En vez de ello las señales *T* e *I* se definen como dos señales de dos bits, las cuales se usan en las instrucciones CASE. El código establece *T* con el valor de *Count*, mientras que *I* se establece con el valor de los dos bits del extremo izquierdo del registro de funciones, que corresponde a los valores almacenados de la entrada *F*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE work.subccts.all ;

ENTITY proc IS
    PORT ( Data      : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Reset, w  : IN     STD_LOGIC ;
           Clock     : IN     STD_LOGIC ;
           F, Rx, Ry : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           Done      : BUFFER STD_LOGIC ;
           BusWires  : INOUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END proc ;

ARCHITECTURE Behavior OF proc IS
    SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL T, I, X, Y : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    decT: dec2to4 PORT MAP ( Count, High, T );
    Func <= F & Rx & Ry ;
    FRin <= w AND T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                 PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    decI: dec2to4 PORT MAP ( FuncReg(1 TO 2), High, I ) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;
    Extern <= I(0) AND T(1) ;
    Done <= ((I(0) OR I(1)) AND T(1)) OR ((I(2) OR I(3)) AND T(3)) ;
    Ain <= (I(2) OR I(3)) AND T(1) ;
    Gin <= (I(2) OR I(3)) AND T(2) ;
    Gout <= (I(2) OR I(3)) AND T(3) ;
    AddSub <= I(3) ;
```

. . . continúa en el inciso *b*.

Figura 7.73 Código para el procesador (inciso *a*).

```
    RegCntl:
    FOR k IN 0 TO 3 GENERATE
        Rin(k) <= ((I(0) OR I(1)) AND T(1) AND X(k)) OR
            ((I(2) OR I(3)) AND T(3) AND X(k)) ;
        Rout(k) <= (I(1) AND T(1) AND Y(k)) OR
            ((I(2) OR I(3)) AND ((T(1) AND X(k)) OR (T(2) AND Y(k)))) ;
    END GENERATE RegCntl ;
    tri_extern: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri0: trin PORT MAP ( R0, Rout(0), BusWires ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu:
    WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0',
               A - BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    triG: trin PORT MAP ( G, Gout, BusWires ) ;
END Behavior ;
```

Figura 7.73 Código para el procesador (inciso *b*).

Hay dos niveles anidados de instrucciones CASE. En el primero se enumeran los valores posibles de *T*. Por cada cláusula WHEN de esta instrucción CASE, la cual representa una columna de la tabla 7.3, hay una instrucción CASE anidada que enumera los cuatro valores de *I*. Según indican los comentarios del código, las instrucciones CASE anidadas corresponden exactamente a la información de la tabla 7.3.

Al final de la figura 7.74, el bus se describe con una asignación de señal seleccionada (Sel). Esta instrucción representa multiplexores que colocan los datos apropiados en *BusWires*, según los valores de R_{out}, G_{out} y *Extern*.

Los circuitos sintetizados a partir del código de las figuras 7.73 y 7.74 son funcionalmente equivalentes. El estilo del código de la figura 7.74 tiene la ventaja de que no requiere el esfuerzo manual de analizar la tabla 7.3 a fin de generar las expresiones lógicas para las señales de control usadas en la figura 7.73. Al emplear el estilo de código de la figura 7.74, estas expresiones se producen en forma automática mediante el compilador de VHDL como resultado del análisis de las instrucciones CASE. El estilo del código de la figura 7.74 es menos propenso a errores. Además, si este estilo se emplea es más sencillo proporcionar capacidades adicionales en el procesador, como la adición de otras operaciones.

```
ARCHITECTURE Behavior OF proc IS
    SIGNAL X, Y, Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero, T, I : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg, Sel : STD_LOGIC_VECTOR(1 TO 6) ;
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND NOT T(1) AND NOT T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    T <= Count ;
    Func <= F & Rx & Ry ;
    FRin <= w AND NOT T(1) AND NOT T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                 PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    I <= FuncReg(1 TO 2) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;

    controlsignals: PROCESS ( T, I, X, Y )
    BEGIN
        Extern <= '0' ; Done <= '0' ; Ain <= '0' ; Gin <= '0' ;
        Gout <= '0' ; AddSub <= '0' ; Rin <= "0000" ; Rout <= "0000" ;
        CASE T IS    WHEN "00" => -- ninguna señal se valida en la etapa de tiempo T0
            WHEN "01" => -- define las señales validadas en la etapa de tiempo T1
                CASE I IS
                    WHEN "00" => -- Load
                        Extern <= '1' ; Rin <= X ; Done <= '1' ;
                    WHEN "01" => -- Move
                        Rout <= Y ; Rin <= X ; Done <= '1' ;
                    WHEN OTHERS => -- Add, Sub
                        Rout <= X ; Ain <= '1' ;
                END CASE ;
```

. . . continúa en el inciso *b*.

Figura 7.74 Código alternativo para el procesador (inciso *a*).

Sintetizamos un circuito para implementar el código de la figura 7.74 en un chip. En la figura 7.75 se da un ejemplo de los resultados de una simulación de tiempo. Cada ciclo del reloj en el que $w = 1$ en este diagrama de tiempo indica el comienzo de una operación. En la primera de estas operaciones, a 250 ns en el tiempo de simulación, los valores de las entradas *F* y *Rx* son 00. Por tanto, la operación corresponde a "*Load R*0,*Data*". El valor de *Data* es 2A, el cual se carga

```
            WHEN ”10” => -- define las señales validadas en la etapa de tiempo T2
                CASE I IS
                    WHEN ”10” => -- Add
                        Rout <= Y ; Gin <= ’1’ ;
                    WHEN ”11” => -- Sub
                        Rout <= Y ; AddSub <= ’1’ ; Gin <= ’1’ ;
                    WHEN OTHERS => -- Load, Move
                END CASE ;
            WHEN OTHERS => -- define las señales validadas en la etapa de tiempo T3
                CASE I IS
                    WHEN ”00” => -- Load
                    WHEN ”01” => -- Move
                    WHEN OTHERS => -- Add, Sub
                        Gout <= ’1’ ; Rin <= X ; Done <= ’1’ ;
                END CASE ;
        END CASE ;
    END PROCESS ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu: WITH AddSub SELECT
        Sum <= A + BusWires WHEN ’0’,
                A − BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    Sel <= Rout & Gout & Extern ;
    WITH Sel SELECT
        BusWires <= R0 WHEN ”100000”,
                    R1 WHEN ”010000”,
                    R2 WHEN ”001000”,
                    R3 WHEN ”000100”,
                    G WHEN  ”000010”,
                    Data WHEN OTHERS ;
END Behavior ;
```

Figura 7.74 Código opcional para el procesador (inciso *b*).

en *R*0 en el siguiente flanco positivo del reloj. La operación siguiente carga 55 en el registro *R*1 y la operación subsiguiente carga 22 en *R*2. A 850 ns el valor de la entrada *F* es 10, mientras que *Rx* = 01 y *Ry* = 00. Esta operación es “*Add R*1,*R*0”. En el ciclo del reloj siguiente, el contenido de *R*1 (55) aparece en el bus. Estos datos se cargan en el registro A por el flanco del reloj a 950 ns, lo cual también da como resultado que el contenido de *R*0 (2A) se coloque en el bus. El módulo sumador/restador genera la suma correcta (7F), la cual se carga en el registro *G* a 1050 ns.

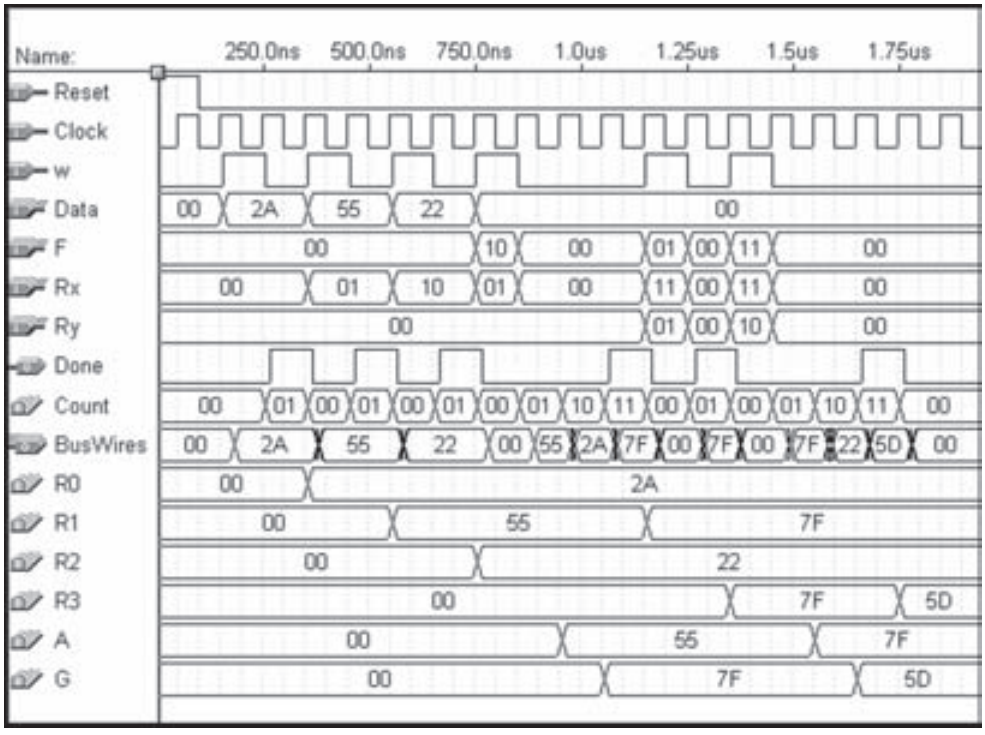


Figura 7.75 Simulación de tiempo para el código de VHDL de la figura 7.74.

Después de este flanco del reloj el contenido nuevo de *G* (7F) se coloca en el bus y se carga en el registro *R*1 a 1150 ns. Dos operaciones más se muestran en el diagrama de tiempo. Una a 1250 ns ("*Move R*3,*R*l") copia el contenido de *R*1 (7F) en *R*3. Finalmente, la operación que empieza en 1450 ns ("*Sub R*3,*R*2") resta el contenido de *R*2 (22) del contenido de *R*3 (7F), lo que produce el resultado correcto, $7F = 22 = 5D$.

7.14.3 CONTADOR DE TIEMPO DE REACCIÓN

En el capítulo 3 mostramos que los dispositivos electrónicos operan a velocidades increíblemente rápidas, con el retraso típico a través de una compuerta lógica menor de 1 ns. En este ejemplo usamos un circuito lógico para medir la velocidad de un tipo de dispositivo más lento: una persona.

Diseñaremos un circuito que puede emplearse para medir el tiempo de reacción de una persona ante un suceso específico. El circuito enciende una pequeña luz, llamada *diodo emisor de luz* (LED, *light-emitting diode*). En respuesta al encendido del LED, la persona intenta oprimir un interruptor lo más rápido posible. El circuito mide el tiempo transcurrido desde el momento en que el LED se encendió hasta que el interruptor se oprime.

Para medir el tiempo de reacción se necesita una señal de reloj con una frecuencia apropiada. En este ejemplo usamos un reloj de 100 Hz, que mide el tiempo en una resolución de 1/100

de segundo. El tiempo de reacción entonces puede exhibirse usando dos dígitos que representan fracciones de segundo desde 00/100 hasta 99/100.

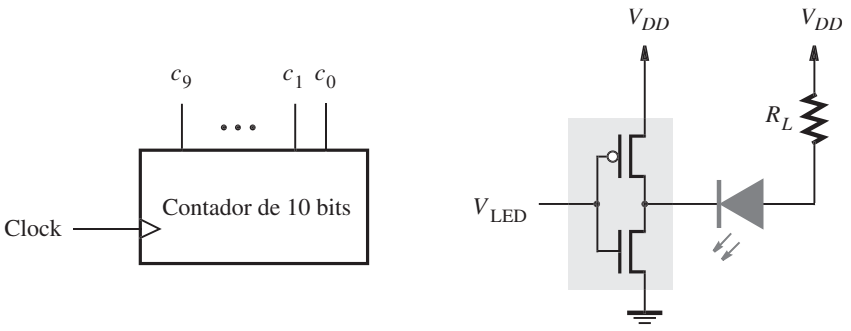
Los sistemas digitales suelen incluir señales de reloj para controlar varios subsistemas. En este caso suponemos la existencia de una señal de reloj de entrada con la frecuencia 102.4 kHz. Desde esta señal podemos derivar la señal de 100 Hz requerida si utilizamos un contador como un *divisor de reloj*. En la figura 7.22 se presenta un diagrama de tiempo para un contador de cuatro bits. Muestra que la salida del bit menos significativo, Q_0, del contador es una señal periódica con la mitad de la frecuencia de la entrada del reloj. Por tanto, podemos considerar que Q_0 divide la frecuencia del reloj entre dos. De forma similar, la salida Q_1 divide la frecuencia del reloj entre cuatro. En general, la salida Q_i en un contador de n bits divide la frecuencia del reloj entre 2^{i+1}. En el caso de nuestra señal de reloj de 102.4 kHz, podemos usar un contador de 10 bits, como se muestra en la figura 7.76*a*. La salida del contador c_9 tiene la frecuencia de 100 Hz requerida porque 102400 Hz/1024 = 100 Hz.

El circuito del contador de tiempo de reacción debe ser capaz de encender un LED y apagarlo. En la figura 7.76*b* se muestra en gris oscuro el símbolo gráfico de un LED. Las flechitas gris oscuro del símbolo representan la luz emitida cuando el LED está encendido. El LED tiene dos terminales: la que está a la izquierda de la figura es el *cátodo*, y la de la derecha es el *ánodo*. Para encender el LED, el cátodo debe establecerse en un voltaje inferior que el ánodo, lo que ocasiona que una corriente fluya por el LED. Si los voltajes en sus dos terminales son iguales, el LED está apagado.

En la figura 7.76*b* se muestra una manera de controlar el LED con un inversor. Si el voltaje de entrada $V_{LED} = 0$, entonces el voltaje en el cátodo es igual a V_{DD}; por tanto, el LED está apagado. Pero si $V_{LED} = V_{DD}$, el voltaje del cátodo es 0 V y el LED está encendido. La cantidad de corriente que fluye está limitada por el valor de la resistencia R_L. Esta corriente pasa por el LED en el transistor NMOS del inversor. Como la corriente fluye *hacia* el inversor, decimos que el inversor *hunde* la corriente. La corriente máxima que una compuerta lógica puede hundir sin sufrir un daño por lo general se llama I_{OL}, que significa "la corriente máxima cuando la salida está en un nivel bajo". El valor de R_L se elige de modo que la corriente sea menor que I_{OL}. Como ejemplo supóngase que el inversor se implementó dentro de un PLD. El valor común de I_{OL}, el cual se especificaría en la hoja de datos del PLD, se aproxima a 12 mA. Para $V_{DD} = 5$ V, esto conduce a $R_L \approx 450\ \Omega$ porque 5 V /450 Ω = 11 mA (en realidad hay una pequeña caída de voltaje en el LED cuando se enciende, pero la ignoramos en aras de la sencillez). La cantidad de luz emitida por el LED es proporcional al flujo de corriente. Si 11 mA es insuficiente, entonces el inversor debe implementarse en un chip de buffer como los descritos en la sección 3.5, ya que los buffers ofrecen un volumen mayor de I_{OL}.

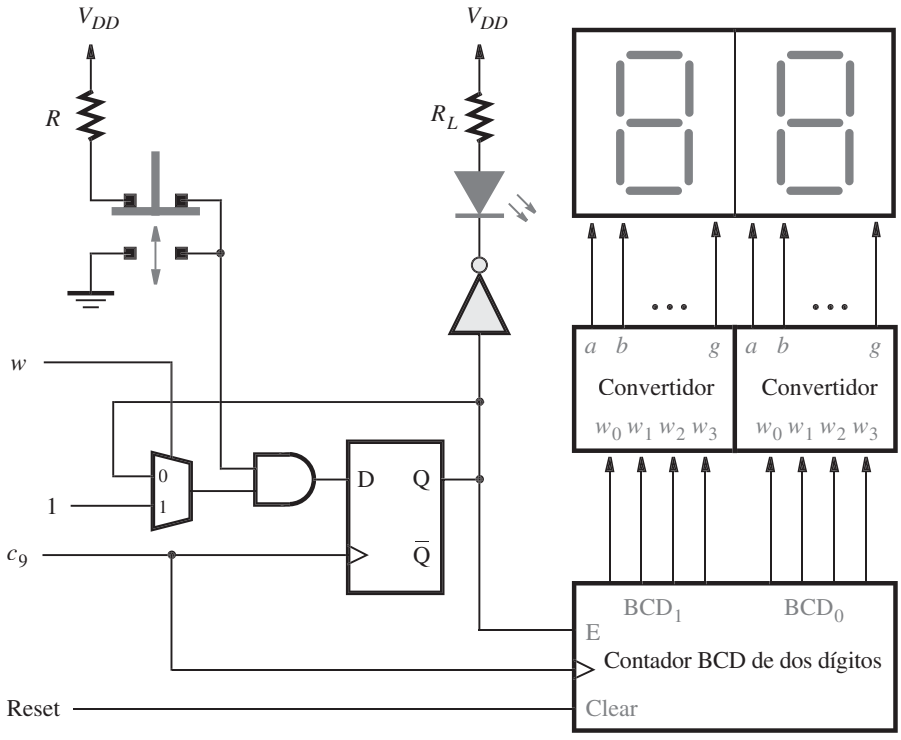
El circuito completo del contador de tiempo de reacción se ilustra en la figura 7.76*c*, con el inversor del inciso *b*) sombreado en gris claro. El símbolo gráfico de un interruptor con un botón para oprimir se muestra a la izquierda del diagrama. El interruptor normalmente hace contacto con las terminales superiores, como se describe en la figura. Cuando se oprime, el interruptor hace contacto con las terminales inferiores; cuando se suelta, automáticamente regresa a la posición superior. En la figura el interruptor está conectado de tal forma que normalmente produce un valor lógico de 1 y un pulso de 0 cuando se presiona.

Oprimir el botón del interruptor causa que el flip-flop D se inicialice de manera síncrona. La salida de este flip-flop determina si el LED está encendido o apagado, y también proporciona la entrada de habilitación del conteo para un contador BCD de dos dígitos. Como estudiamos en la sección 7.11, cada dígito en un contador BCD tiene cuatro bits que toman los valores 0000 a 1001.



a) Divisor del reloj

b) Circuito LED



c) Interruptor de botón, LED, y pantallas de siete segmentos

Figura 7.76 Un circuito contador de tiempo de reacción.

Por tanto, la secuencia de conteo puede verse como números decimales desde 00 a 99. Un circuito para el contador BCD se da en la figura 7.28. En la figura 7.76*c* tanto el flip-flop como el contador se sincronizan por la salida c_9 del divisor del reloj en el inciso *a*) de la figura. El uso buscado del circuito contador de tiempo de reacción consiste en primero oprimir el interruptor para apagar el LED e inhabilitar el contador. Luego la entrada *Reset* se valida para borrar el contenido del contador y dejarlo en 00. La entrada *w* normalmente tiene el valor 0, el cual mantiene el flip-flop borrado e impide que cambie el valor de conteo. La prueba de reacción se inicia al establecer $w = 1$ para un ciclo del reloj c_9. Después del siguiente flanco positivo de c_9, la salida del flip-flop se vuelve 1, con lo que el LED se enciende. Suponemos que *w* vuelve a 0 después de un ciclo del reloj, pero la salida del flip-flop permanece en 1 porque el multiplexor dos a uno está conectado a la entrada D. El contador entonces se incrementa cada 1/100 de segundo. Cada dígito del contador se conecta por medio de un convertidor de código a una pantalla de siete segmentos, la cual se describió en el análisis de la figura 6.25. Cuando el usuario presiona el interruptor, el flip-flop se borra, con lo que el LED se apaga y el contador se detiene. La pantalla de dos dígitos muestra el tiempo transcurrido hasta el 1/100 de segundo más cercano desde que el LED se encendió hasta que el usuario pudo responder oprimiendo el interruptor.

Código de VHDL

Para describir el circuito de la figura 7.76*c* con código de VHDL podemos emplear los subcircuitos para el contador BCD y el código para el convertidor de siete segmentos. Este último se dio en la figura 6.47 y no lo repetiremos aquí. El código para el contador BCD, que representa el circuito de la figura 7.28, se muestra en la figura 7.77. La salida BCD de dos dígitos está representada por las dos señales de cuatro dígitos *BCD*1 y *BCD*0. La entrada *Clear* sirve para proporcionar un reset síncrono para los dos dígitos en el contador. Si $E = 1$, el valor de conteo se incrementa en el flanco positivo del reloj, y si $E = 0$ el valor de conteo no sufre cambio alguno. Cada dígito puede tomar los valores de 0000 a 1001.

En la figura 7.78 se proporciona el código para el contador de tiempo de reacción. La señal de entrada *Pushn* representa el valor producido por el interruptor de botón para oprimir. La señal de salida *LEDn* representa la salida del inversor que se usa para controlar el LED. Las dos pantallas de siete segmentos están controladas por las señales de siete bits *Digit*1 y *Digit*0.

En la figura 7.56 se muestra cómo un registro, *R*, puede diseñarse con una señal de control R_{in}. Si $R_{in} = 1$ los datos se cargan en el registro en el flanco activo del reloj y si $R_{in} = 0$ el contenido almacenado del registro no cambia. El flip-flop de la figura 7.76 se usa de la misma forma. Si $w = 1$, el flip-flop se carga con el valor 1, pero si $w = 0$ el valor almacenado en el flip-flop no cambia. Este circuito se describe mediante el proceso etiquetado *flipflop* de la figura 7.78, que también incluye una entrada de reset síncrono. Hemos optado por usar un reset síncrono porque la salida del flip-flop está conectada a la entrada de habilitación *E* en el contador BCD. Como sabemos a partir de lo explicado en la sección 7.3, es importante que todas las señales conectadas a los flip-flops satisfagan los tiempos de preparación y espera requeridos. El interruptor de botón puede presionarse en cualquier momento y no está sincronizado por la señal c_9 del reloj. Al usar un reset síncrono para el flip-flop de la figura 7.76, evitamos problemas de sincronización potenciales en el contador.

La salida del flip-flop se denomina *LED*, la cual se invierte para producir la señal *LEDn* que controla el LED. En el dispositivo usado para implementar el circuito, *LEDn* sería generada por el buffer que está conectado a un pin de salida en el paquete de chips. Si se emplea un PLD, este buffer tiene el valor asociado de $I_{OL} = 12$ mA que mencionamos antes. Al final de la figura 7.78 el contador BCD y los convertidores de siete segmentos se instancian como subcircuitos.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCDcount IS
    PORT ( Clock          : IN     STD_LOGIC ;
           Clear, E       : IN     STD_LOGIC ;
           BCD1, BCD0     : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END BCDcount ;

ARCHITECTURE Behavior OF BCDcount IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Clear = '1' THEN
                BCD1 <= "0000" ; BCD0 <= "0000" ;
            ELSIF E = '1' THEN
                IF BCD0 = "1001" THEN
                    BCD0 <= "0000" ;
                    IF BCD1 = "1001" THEN
                        BCD1 <= "0000";
                    ELSE
                        BCD1 <= BCD1 + '1' ;
                    END IF ;
                ELSE
                    BCD0 <= BCD0 + '1' ;
                END IF ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Figura 7.77 Código para el contador BCD de dos dígitos de la figura 7.28.

Una simulación del circuito contador de tiempo de reacción implementado en un chip se muestra en la figura 7.79. Al principio, *Pushn* se establece en 0 para simular que se oprime el interruptor de modo que se encienda el LED y luego *Pushn* regresa a 1. Además, *Reset* se valida para borrar el contador. Cuando *w* cambia a 1, el circuito establce *LEDn* en 0, lo que representa el encendido del LED. Después de un tiempo el interruptor se presionará. En la simulación establecimos arbitrariamente *Pushn* en 0 después de 18 ciclos del reloj c_9. Por tanto, esta elección representa el caso en que el tiempo de reacción de la persona es de alrededor de 0.18 segundos. En términos humanos, esta duración es muy breve; para los circuitos electrónicos es un tiempo muy largo. ¡Una computadora personal barata puede realizar decenas de millones de operaciones en 0.18 segundos!

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reaction IS
    PORT ( c9, Reset        : IN        STD_LOGIC ;
           w, Pushn         : IN        STD_LOGIC ;
           LEDn             : OUT       STD_LOGIC ;
           Digit1, Digit0   : BUFFER    STD_LOGIC_VECTOR(1 TO 7) ) ;
END reaction ;

ARCHITECTURE Behavior OF reaction IS
    COMPONENT BCDcount
        PORT ( Clock            : IN        STD_LOGIC ;
               Clear, E         : IN        STD_LOGIC ;
               BCD1, BCD0       : BUFFER    STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT seg7
        PORT ( bcd    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               leds   : OUT    STD_LOGIC_VECTOR(1 TO 7) ) ;
    END COMPONENT ;
    SIGNAL LED : STD_LOGIC ;
    SIGNAL BCD1, BCD0 : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
BEGIN
    flipflop: PROCESS
    BEGIN
        WAIT UNTIL c9'EVENT AND c9 = '1' ;
        IF Pushn = '0' THEN
            LED <= '0' ;
        ELSIF w = '1' THEN
            LED <= '1' ;
        END IF ;
    END PROCESS ;

    LEDn <= NOT LED ;
    counter: BCDcount PORT MAP ( c9, Reset, LED, BCD1, BCD0 ) ;
    seg1 : seg7 PORT MAP ( BCD1, Digit1 ) ;
    seg0 : seg7 PORT MAP ( BCD0, Digit0 ) ;
END Behavior ;
```

Figura 7.78 Código para el contador de tiempo de reacción.

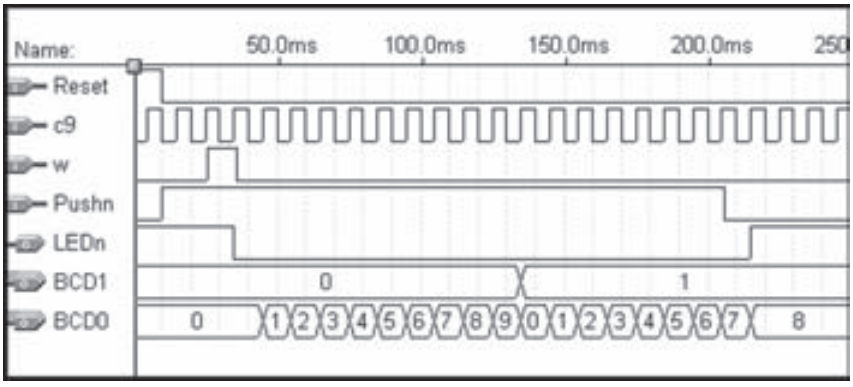


Figura 7.79 Simulación del circuito contador de tiempo de reacción.

7.14.4 CÓDIGO DE NIVEL DE TRANSFERENCIA DE REGISTROS (RTL)

Hasta ahora hemos presentado la mayor parte de los constructores de VHDL necesarios para la síntesis. Casi todos nuestros ejemplos muestran código por comportamiento que utiliza instrucciones IF-THEN-ELSE, CASE, FOR LOOP, etcétera. Es posible escribir código por comportamiento en un estilo parecido a un programa de computadora, en el cual hay un flujo de control complejo con muchos ciclos y ramas. Con un código como ése, a veces llamado código por comportamiento de *alto nivel*, es difícil relacionar el código con la implementación del hardware final; incluso podría ser difícil predecir qué circuito producirá una herramienta de síntesis de alto nivel. En este libro no usamos el estilo de código de alto nivel. En vez de ello, presentamos el código de VHDL de forma tal que pueda relacionarse fácilmente con el circuito que se está describiendo. El grueso de los módulos de diseño presentados es muy pequeño a fin de ofrecer descripciones sencillas. Los diseños más grandes se construyen interconectando los módulos más pequeños. Este enfoque se conoce como estilo de código de *nivel de transferencia de registros* (RTL, *register-transfer level*). Es el enfoque de diseño más popular en la práctica. El código RTL se caracteriza por un flujo de control directo a través del código; comprende subcircuitos bien entendidos y conectados juntos de una manera simple.

7.15 COMENTARIOS FINALES

En este capítulo hemos presentado circuitos que sirven como elementos de almacenamiento básicos en los sistemas digitales. Estos elementos se utilizan para construir unidades más grandes, como registros, registros de corrimiento y contadores. Muchos otros libros abordan esos temas [3-11]. Hemos ilustrado cómo los circuitos con flip-flops pueden describirse con código de VHDL. Más información respecto a VHDL puede encontrarse en [12-17]. En el capítulo siguiente se presentará un método más formal para diseñar circuitos con flip-flops.

7.16 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas comunes que el lector puede encontrar y se muestra cómo resolverlos.

Ejemplo 7.13

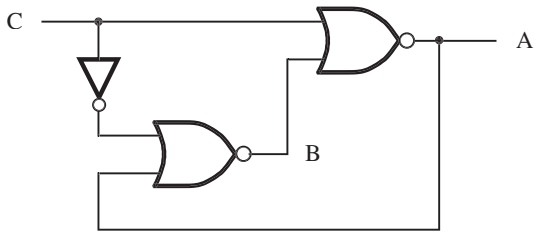
Problema: Considere el circuito de la figura 7.80*a*. Suponga que la entrada *C* está manejada por una señal de onda cuadrada con un ciclo de trabajo de 50%. Dibuje un diagrama de tiempo que muestre las formas de onda en los puntos *A* y *B*. Suponga que el retraso de propagación por cada compuerta es de Δ segundos.

Solución: El diagrama de tiempo se muestra en la figura 7.80*b*.

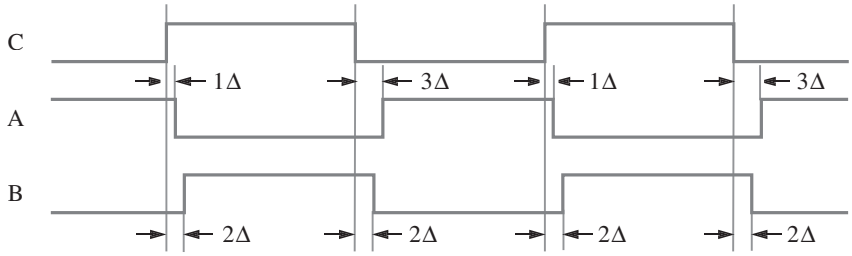
Ejemplo 7.14

Problema: Determine el comportamiento funcional del circuito de la figura 7.81. Suponga que la entrada *w* está manejada por una señal de onda cuadrada.

Solución: Cuando los dos flip-flops se borran, sus salidas son $Q_0 = Q_1 = 0$. Después que la entrada Clear adquiere un nivel alto, cada pulso en la entrada *w* ocasionará un cambio en los



a) Circuito



b) Diagrama de tiempo

Figura 7.80 Circuito para el ejemplo 7.13.

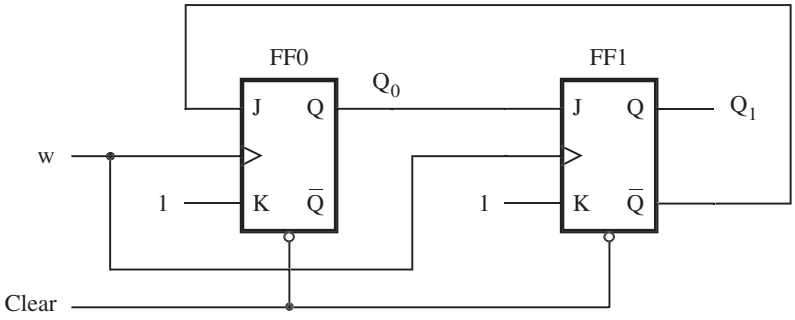


Figura 7.81 Circuito para el ejemplo 7.14.

Intervalo de tiempo	FF0			FF1		
	J_0	K_0	Q_0	J_1	K_1	Q_1
Clear	1	1	0	0	1	0
t_1	1	1	1	1	1	0
t_2	0	1	0	0	1	1
t_3	1	1	0	0	1	0
t_4	1	1	1	1	1	0

Figura 7.82 Resumen del comportamiento del circuito de la figura 7.81.

flip-flops según se indica en la figura 7.82. Observe que la figura muestra el estado de las señales después de los cambios ocasionados por el flanco de subida de un pulso.

En los intervalos de tiempo consecutivos los valores de Q_1 Q_0 son 00, 01, 10, 00, 01 y así sucesivamente. Por consiguiente, el circuito genera la secuencia de conteo 0, 1, 2, 0, 1 etcétera. Por tanto, el circuito es un contador módulo 3.

Ejemplo 7.15 **Problema:** En la figura 7.70 se muestra un circuito que genera cuatro señales de control de tiempo T_0, T_1, T_2 y T_3. Diseñe un circuito que genere seis señales como éstas, T_0 a T_5.

Solución: El esquema de la figura 7.70 puede ampliarse al usar un contador módulo 6, dado en la figura 7.26, y un decodificador que produce las seis señales de sincronización. Una alternativa más simple es usar un contador Johnson. Si empleamos tres flip-flops tipo D en una estructura como la descrita en la figura 7.30, podemos generar seis patrones de bits $Q_0Q_1Q_2$ como se muestra en la figura 7.83. Luego, usando otras seis compuertas AND de dos entradas, como se advierte en la figura, podemos obtener las señales buscadas. Note que los patrones $Q_0Q_1Q_2$ iguales a 010 y 101 no pueden ocurrir en el contador Johnson, así que los casos se tratan como condiciones sin importancia.

Ciclo del reloj	Q_0	Q_1	Q_2	Señal de control
0	0	0	0	$T_0 = \overline{Q}_0\overline{Q}_2$
1	1	0	0	$T_1 = Q_0\overline{Q}_1$
2	1	1	0	$T_2 = Q_1\overline{Q}_2$
3	1	1	1	$T_3 = Q_0Q_2$
4	0	1	1	$T_4 = \overline{Q}_0Q_1$
5	0	0	1	$T_5 = \overline{Q}_1Q_2$

Figura 7.83 Señales de tiempo para el ejemplo 7.15.

Ejemplo 7.16

Problema: Diseñe un circuito que sirva para controlar una máquina expendedora. El circuito tiene cinco entradas: Q (moneda de 25 centavos), D (moneda de 10 centavos), N (moneda de 5 centavos), *Coin* y *Resetn*. Cuando una moneda se deposita en la máquina, un mecanismo detector de monedas genera un pulso en la entrada apropiada (Q, D o N). Para dar significado a la ocurrencia del evento, el mecanismo también genera un pulso en la línea *Coin*. El circuito se inicializa usando la señal *Resetn* (activa en nivel bajo). Cuando se han depositado al menos 30 centavos, el circuito activa su salida, Z. No se da cambio si el monto excede 30 centavos.

Diseñe el circuito usando los componentes siguientes: un sumador de seis bits, un registro de seis bits y cualquier número de compuertas AND, OR y NOT.

Solución: En la figura 7.84 se muestra un circuito posible. El valor de cada moneda se representa por medio de un número correspondiente de cinco bits. Se añade al total actual, el cual se aloja en el registro *S*. La salida requerida es

$$Z = s_5 + s_4s_3s_2s_1$$

El registro se sincroniza por el flanco negativo de la señal *Coin*. Esto permite un retraso de propagación a través del sumador y asegura que una suma correcta se coloque en el registro.

En el capítulo 9 mostraremos cómo este tipo de circuito de control puede diseñarse usando un enfoque más estructurado.

Ejemplo 7.17

Problema: Escriba código de VHDL para implementar el circuito de la figura 7.84.

Solución: En la figura 7.85 se proporciona el código deseado.

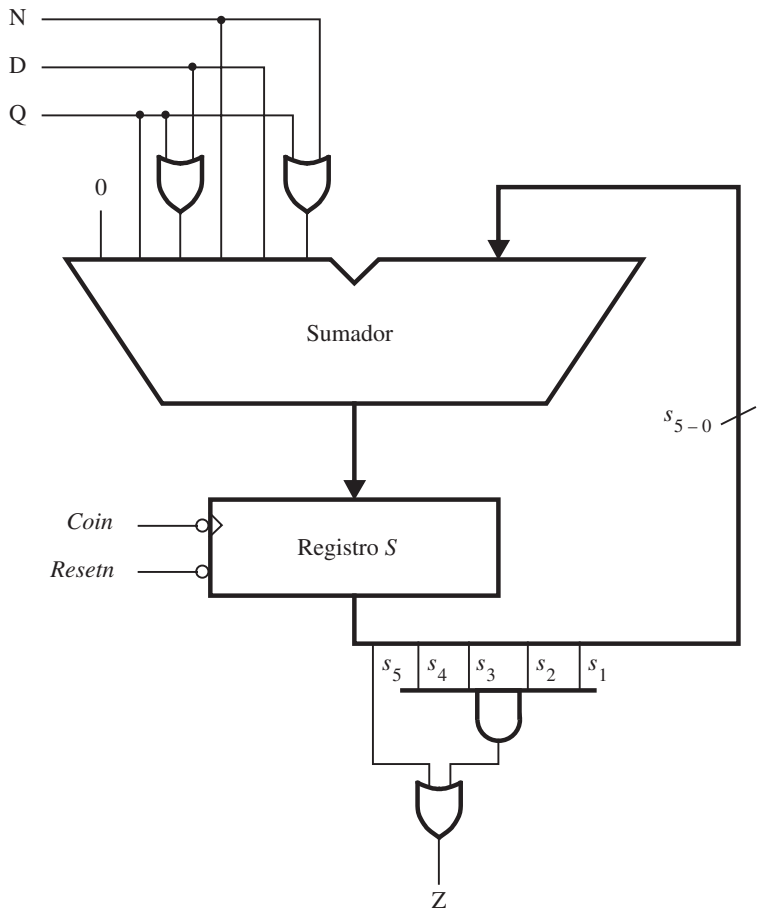


Figura 7.84 Circuito para el ejemplo 7.16.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY vend IS
    PORT ( N, D, Q, Resetn, Coin  : IN   STD_LOGIC ;
           Z                      : OUT  STD_LOGIC ) ;
END vend ;

ARCHITECTURE Behavior OF vend IS
    SIGNAL X:  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL S:  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
BEGIN
    X(0) <= N OR Q ;
    X(1) <= D ;
    X(2) <= N ;
    X(3) <= D OR Q ;
    X(4) <= Q ;
    PROCESS ( Resetn, Coin )
    BEGIN
        IF Resetn = '0' THEN
            S <= "000000" ;
        ELSIF Coin'EVENT AND Coin = '0' THEN
            S <= ('0' & X) + S ;
        END IF ;
    END PROCESS ;
    Z <= S(5) OR (S(4) AND S(3) AND S(2) AND S(1)) ;
END Behavior ;
```

Figura 7.85 Código para el ejemplo 7.17.

PROBLEMAS

Al final del libro se incluyen las respuestas a los problemas marcados con asterisco.

7.1 Considere el diagrama de tiempo de la figura P7.1. Suponiendo que las entradas *D* y *Clock* mostradas se aplican al circuito de la figura 7.12, dibuje formas de onda para las señales Q_a, Q_b y Q_c.

7.2 ¿El circuito de la figura 7.3 puede modificarse para implementar un latch SR? Explique su respuesta.

7.3 En la figura 7.5 se muestra un latch construido con compuertas NOR. Dibuje un latch parecido usando compuertas NAND. Derive su tabla característica y muestre su diagrama de tiempo.

***7.4** Muestre un circuito que implemente el latch SR asíncrono usando únicamente compuertas NAND.

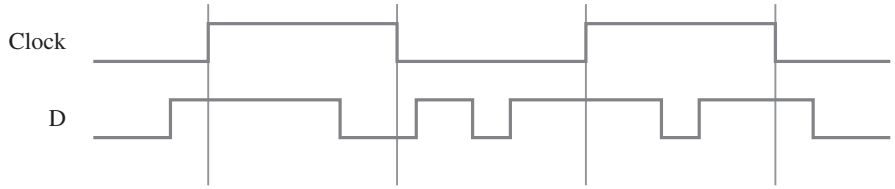


Figura P7.1 Diagrama de tiempo para el problema 7.1.

7.5 Dada una señal de reloj de 100 MHz, derive un circuito usando flip-flops D para generar señales de reloj de 50 y 25 MHz. Trace un diagrama de tiempo para las tres señales de reloj, suponiendo retrasos razonables.

***7.6** Un flip-flop SR es uno que tiene entradas set y reset como un latch SR asíncrono. Muestre cómo puede construirse un flip-flop SR mediante un flip-flop D y otras compuertas lógicas.

7.7 El latch SR asíncrono de la figura 7.6*a* tiene un comportamiento impredecible si las entradas *S* y *R* son iguales a 1 cuando *Clk* cambia a 0. Una forma de resolver este problema es crear un latch SR asíncrono con *set dominante* en el que la condición $S = R = 1$ hace que el latch se establezca en 1. Diseñe un latch SR asíncrono con set dominante y muestre el circuito.

7.8 Muestre cómo un flip-flop JK puede construirse con un flip-flop T y otras compuertas lógicas.

***7.9** Considere el circuito de la figura P7.2. Suponga que las dos compuertas NAND tienen retrasos de propagación mucho más grandes (alrededor de cuatro veces) que las otras compuertas del circuito. ¿Cómo se compara este circuito con los circuitos que hemos estudiado en este capítulo?

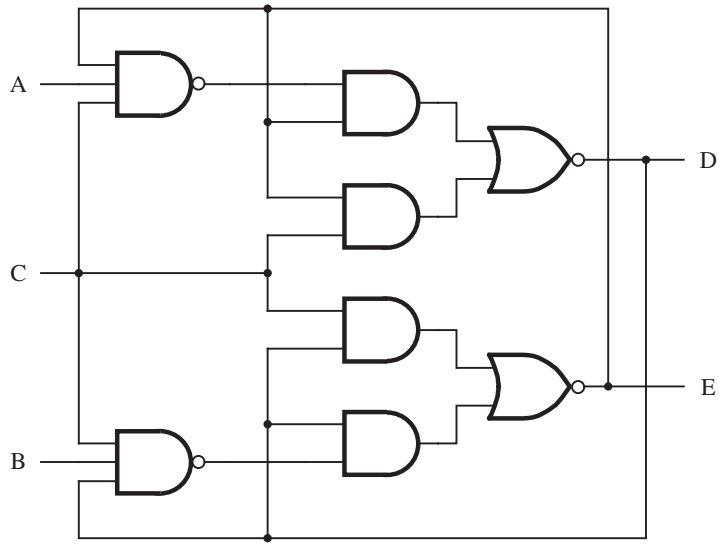


Figura P7.2 Circuito para el problema 7.9.

7.10 Escriba código de VHDL que represente un flip-flop T con una entrada clear asíncrona. Use código por comportamiento en vez de código estructural.

7.11 Escriba código de VHDL que represente un flip-flop JK. Utilice código por comportamiento en vez de código estructural.

7.12 Sintetice un circuito para el código escrito para el problema 7.11 empleando sus herramientas CAD. Simule el circuito y muestre un diagrama de tiempo que verifique la funcionalidad deseada.

7.13 Un registro de corrimiento universal puede desplazarse tanto de izquierda a derecha como de derecha a izquierda, y tiene una capacidad de carga en paralelo. Dibuje un circuito para este registro de corrimiento.

7.14 Escriba código de VHDL para un registro de corrimiento universal con *n* bits.

7.15 Diseñe un contador síncrono de cuatro bits con carga en paralelo. Use flip-flops T en vez de los flip-flops D utilizados en la sección 7.9.3.

***7.16** Diseñe un contador ascendente/descendente de tres bits con flip-flops T. Debe incluir una entrada de control llamada $\overline{\text{Up}}$/Down. Si $\overline{\text{Up}}$/Down $= 0$, entonces el circuito debe comportarse como un contador ascendente. Si es igual a 1, entonces el circuito debe comportarse como un contador descendente.

7.10 Repita el problema 7.16 usando flip-flops D.

***7.18** El circuito de la figura P7.3 parece un contador. ¿Cuál es la secuencia en que cuenta?

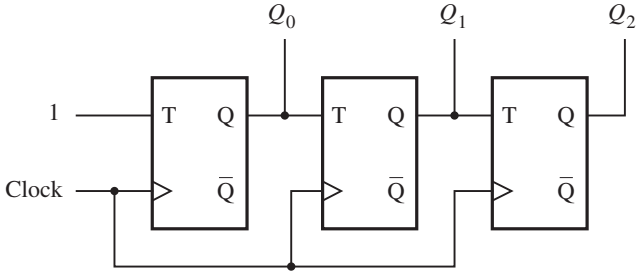


Figura P7.3 Circuito para el problema 7.18.

7.19 Considere el circuito de la figura P7.4. ¿Cómo se compara con el circuito de la figura 7.17? ¿Los dos circuitos pueden usarse con el mismo propósito? Si no es así, ¿cuál es la diferencia principal entre ellos?

7.20 Construya un circuito con compuertas NOR, parecido al de la figura 7.11*a*, el cual implementa un flip-flop D disparado por el flanco negativo.

7.21 Escriba código por comportamiento de VHDL que represente un contador ascendente/descendente de 24 bits con una carga en paralelo y un reset asíncrono.

7.22 Modifique el código de VHDL de la figura 7.52 agregándole un parámetro que establezca el número de flip-flops en el contador.

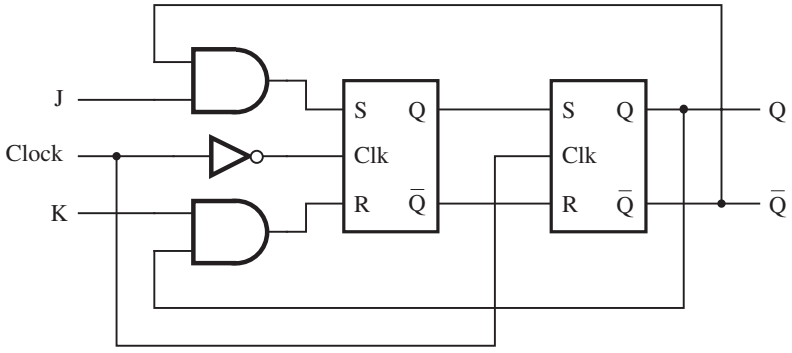


Figura P7.4 Circuito para el problema 7.19.

7.23 Escriba el código por comportamiento de VHDL que represente un contador ascendente módulo 12 con reset síncrono.

***7.28** Para los flip-flops del contador de la figura 7.25, suponga que $t_{su} = 3$ ns, $t_h = 1$ ns y el retraso de propagación a través de un flip-flop es 1 ns. Asuma que cada compuerta AND y XOR y cada multiplexor dos a uno tiene un retraso de propagación igual a 1 ns. ¿Cuál es la frecuencia de reloj máxima para la que el circuito funcionará correctamente?

7.25 Escriba código jerárquico (estructural) para el circuito de la figura 7.28. Use el contador de la figura 7.25 como un subcircuito.

7.26 Escriba código de VHDL que represente un contador Johnson de ocho bits. Sintetice el código con sus herramientas CAD y dé una simulación de tiempo que muestre la secuencia de conteo.

7.27 Escriba código por comportamiento de VHDL en el estilo mostrado en la figura 7.51 que represente un contador en anillo. Su código debe tener un parámetro *N* que establezca el número de flip-flops en el contador.

***7.28** Escriba código por comportamiento de VHDL que describa la funcionalidad del circuito mostrado en la figura 7.42.

7.29 En la figura 7.65 se proporciona un código de VHDL para un sistema digital que intercambia el contenido de dos registros, *R*1 y *R*2, usando el registro *R*3 para almacenamiento temporal. Construya un esquema equivalente con sus herramientas CAD para este sistema. Sintetice un circuito para este esquema y realice una simulación de tiempo.

7.30 Repita el problema 7.29 usando el circuito de control de la figura 7.59.

7.31 Modifique el código de la figura 7.67 para utilizar el circuito de control de la figura 7.59. Sintetice el código para su implementación en un chip y realice una simulación de tiempo.

7.32 En la sección 7.14.2 diseñamos un procesador que efectúa las operaciones indicadas en la tabla 7.3. Diseñe un circuito modificado que realice una operación adicional, Swap Rx, Ry. Esta operación intercambia el contenido de los registros *Rx* y *Ry*. Use tres bits $f_2 f_1 f_0$ para representar la entrada *F* mostrada en la figura 7.71 porque ahora hay cinco operaciones en vez de cuatro. Añada un nuevo registro, llamado *Tmp*, al sistema para que sea el almacenamiento temporal durante la operación de intercambio. Muestre expresiones lógicas para las salidas del circuito de control, como se hizo en la sección 7.14.2.

7.33 Un oscilador en anillo es un circuito que tiene un número impar, n, de inversores conectados en una estructura tipo anillo, como se muestra en la figura P7.5. La salida de cada inversor es una señal periódica con cierto periodo.

a) Suponga que todos los inversores son idénticos; en consecuencia, todos tienen el mismo retraso, llamado t_p. Sea f la salida de uno de los inversores. Dé una ecuación que exprese el periodo de la señal f en términos de n y t_p.

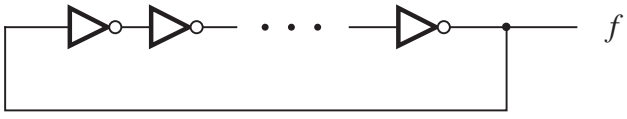


Figura P7.5 Un oscilador en anillo.

b) Para este inciso usted va a diseñar un circuito que pueda usarse para medir en forma experimental el retraso t_p a través de uno de los inversores en el oscilador en anillo. Suponga la existencia de una entrada llamada *Reset* y otra llamada *Interval*. La sincronización de estas dos señales se muestra en la figura P7.6. El periodo para el cual *Interval* tiene el valor 1 se conoce. Suponga que es 100 ns. Diseñe un circuito que utilice las señales *Reset* e *Interval* y la señal f del inciso *a*) para medir experimentalmente t_p. En su diseño puede usar compuertas lógicas y subcircuitos como sumadores, flip-flops, contadores, registros o cualquier otro.

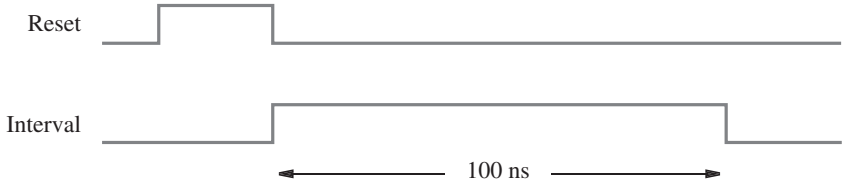


Figura P7.6 Sincronización de las señales para el problema 7.31.

7.34 Un circuito para un latch D asíncrono se muestra en la figura P7.7. Suponga que el retraso de propagación a través de una compuerta NAND o de un inversor es de 1 ns. Complete el diagrama de tiempo dado en la figura, el cual muestra los valores de la señal con resolución de 1 ns.

***7.35** Un circuito lógico tiene dos entradas, *Clock* y *Start*, y dos salidas, f y g. El comportamiento del circuito se describe en el diagrama de tiempo de la figura P7.8. Cuando se recibe un pulso en la entrada *Start*, el circuito produce pulsos en las salidas f y g como se muestra en el diagrama de tiempo. Diseñe un circuito adecuado usando sólo los componentes siguientes: un contador síncrono de tres bits capaz de inicializarse y disparado por el flanco positivo, y compuertas lógicas básicas. Para dar su respuesta suponga que los retrasos a través de todas las compuertas lógicas y el contador son insignificantes.

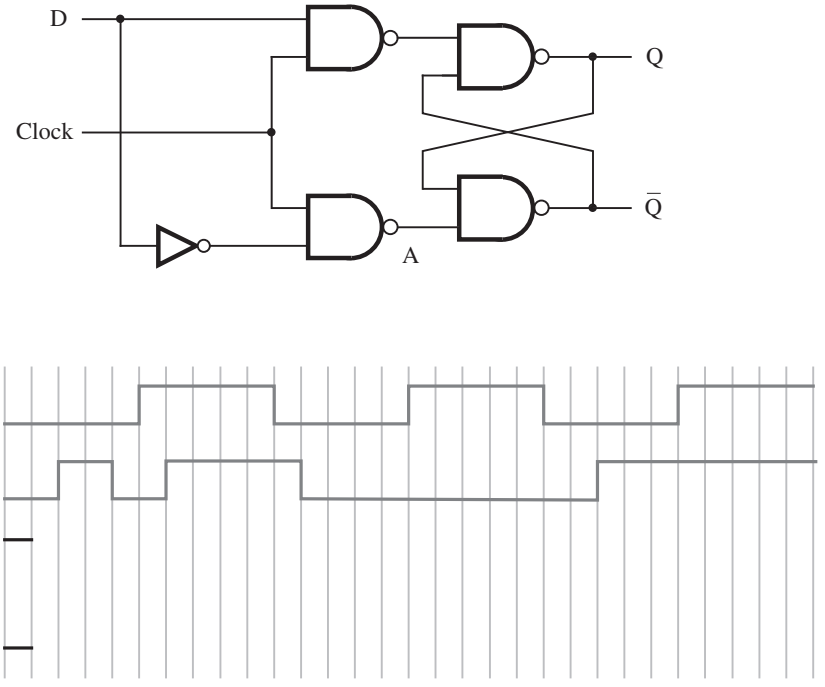


Figura P7.7 Circuito y diagrama de tiempo para el problema 7.32.

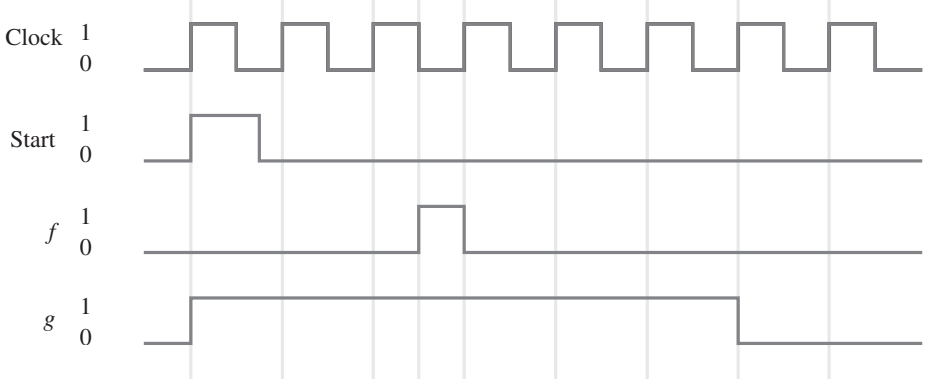


Figura P7.8 Diagrama de tiempo para el problema 7.33.

BIBLIOGRAFÍA

1. V. C. Hamacher, Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 5a. ed. (McGraw-Hill: Nueva York, 2002).
2. D. A. Patterson y J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2a. ed. (Morgan Kaufmann: San Francisco, Ca., 1998).
3. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, N.J., 1997).
4. M. M. Mano, *Digital Design*, 3a. ed. (Prentice-Hall: Upper Saddle River, N.J., 2002).
5. J. P. Daniels, *Digital Design from Zero to One* (Wiley: Nueva York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, N.J., 1995).
7. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, Ca., 1994).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, Ma., 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4a. ed., (West: St. Paul, Mn., 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, 3a. ed. (Prentice-Hall: Englewood Cliffs, N.J., 1999).
11. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, N.J., 1986).
12. Institute of Electrical and Electronics Engineers, "1076-1993 IEEE Standard VHDL Language Reference Manual", 1993.
13. D. L. Perry, *VHDL*, 3a. ed. (McGraw-Hill: Nueva York, 1998).
14. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2a. ed. (McGraw-Hill: Nueva York, 1998).
15. J. Bhasker, *A VHDL Primer*, 3a. ed. (Prentice-Hall: Englewood Cliffs, N.J., 1998).
16. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, Ca., 1996).
17. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, Ma., 1997).

capítulo

8

CIRCUITOS SÍNCRONOS SECUENCIALES

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

- Técnicas de diseño para circuitos que utilizan flip-flops
- El concepto de estados y su implementación con flip-flops
- El control síncrono mediante una señal de reloj
- El comportamiento secuencial de los circuitos digitales
- Un procedimiento completo para diseñar circuitos síncronos secuenciales
- La especificación de VHDL de los circuitos secuenciales
- El concepto de máquinas de estado finito

En capítulos anteriores estudiamos los circuitos lógicos combinacionales cuyas salidas están determinadas por completo por los valores presentes en las entradas. También expusimos cómo pueden implementarse los elementos de almacenamiento simple en forma de flip-flops. La salida de un flip-flop depende de su estado en vez del valor de sus entradas en cualquier momento; las entradas producen cambios en el estado.

En este capítulo abordamos una clase general de circuitos en los que las salidas dependen de la conducta anterior del circuito, así como de los valores presentes en las entradas. Se llaman *circuitos secuenciales*. En la mayor parte de los casos se usa una señal de reloj para controlar la operación de un circuito secuencial; un circuito de este tipo se llama *circuito secuencial síncrono*. La alternativa, en la que no se emplea ninguna señal de reloj, se denomina *circuito secuencial asíncrono*. Es más fácil diseñar circuitos síncronos y se utilizan en la inmensa mayoría de las aplicaciones prácticas. Estos circuitos son el tema del capítulo presente. Los circuitos asíncronos se estudian en el capítulo siguiente.

Los circuitos síncronos secuenciales se realizan usando la lógica combinacional y uno o más flip-flops. Su estructura general se muestra en la figura 8.1. El circuito tiene una serie de entradas principales, *W*, y produce una serie de salidas, *Z*. Los valores de las salidas de los flip-flops se conocen como el *estado*, Q, del circuito. Bajo el control de la señal de reloj, las salidas de los flip-flops cambian su estado según lo determina la lógica combinacional que alimenta las entradas de esos flip-flops. De esta manera el circuito pasa de un estado a otro. Para asegurar que sólo hay una transición de un estado a otro durante un ciclo del reloj, los flip-flops deben ser del tipo disparado por flanco. Pueden dispararse ya sea por el flanco positivo (transición de 0 a 1) o por el negativo (transición de 1 a 0) del reloj. Emplearemos el término *flanco activo del reloj* para referirnos al flanco del reloj que produce el cambio de estado.

La lógica combinacional que proporciona las señales de entrada a los flip-flops deriva sus entradas de dos fuentes: las entradas principales, *W*, y las salidas presentes (actuales) de los flip-flops, Q. Por tanto, los cambios de estado dependen tanto del estado presente como de los valores de las entradas principales.

En la figura 8.1 se indica que las salidas del circuito secuencial se generan mediante otro circuito combinacional, de tal forma que las salidas son una función del estado presente de los flip-flops y de las entradas principales. Aun cuando las salidas siempre dependen del estado presente, no necesariamente deben depender de manera directa de las entradas principales. Por ende, la conexión mostrada en gris en la figura podría existir o no. Para distinguir entre estas dos posibilidades, se acostumbra decir que los circuitos secuenciales cuyas salidas dependen sólo del estado del circuito son del tipo *Moore*, mientras que aquellas cuyas salidas dependen tanto del estado como de las entradas principales son del tipo *Mealy*. Estos nombres se asignaron en honor a Edward Moore y George Mealy, quienes investigaron el comportamiento de estos circuitos en la década de 1950.

Los circuitos secuenciales también se llaman *máquinas de estado finito* (FSM, *finite state machines*), un nombre más formal que a menudo se halla en la bibliografía técnica. El nombre proviene del hecho de que el comportamiento funcional de estos circuitos puede representarse mediante un número finito de estados. En este capítulo con frecuencia usamos el término *máquina de estado finito*, o simplemente *máquina*, cuando nos referimos a los circuitos secuenciales.

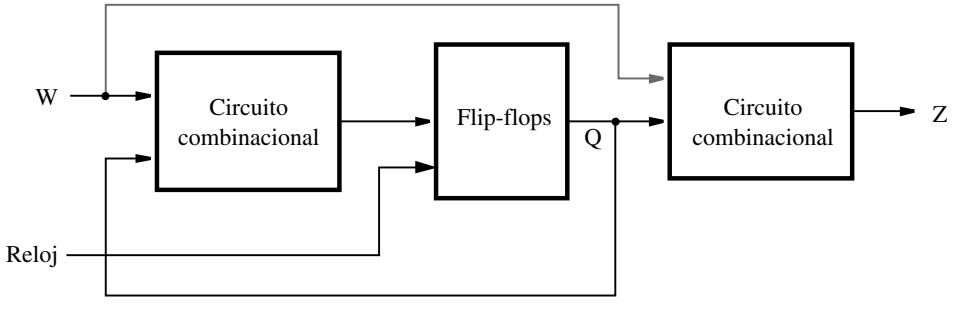


Figura 8.1 La forma general de un circuito secuencial.

8.1 PASOS BÁSICOS DE DISEÑO

Presentaremos las técnicas para el diseño de circuitos secuenciales por medio de un ejemplo sencillo. Supóngase que deseamos diseñar un circuito que cumpla con la especificación siguiente:

1. El circuito tiene una entrada, w, y una salida, z.
2. Todos los cambios en el circuito deben ocurrir en el flanco positivo de una señal de reloj.
3. La salida z es igual a 1 si durante dos ciclos del reloj inmediatamente anteriores la entrada w era igual 1. De lo contrario, el valor de z es igual a 0.

Por tanto, el circuito detecta si dos o más 1 consecutivos ocurren en su entrada w. Los circuitos que detectan la ocurrencia de un patrón en particular en su(s) entrada(s) se conocen como *detectores de secuencia*.

A partir de esta especificación es evidente que la salida z no puede depender únicamente del valor presente en w. Para ilustrar esto, considérese la secuencia de valores de las señales w y z durante 11 ciclos del reloj, como se muestra en la figura 8.2. Los valores de w se suponen arbitrariamente; los valores de z corresponden a nuestra especificación. Estas secuencias de valores de entrada y salida indican que para un valor de entrada, la salida puede ser 0 o 1. Por ejemplo, $w = 0$ durante los ciclos del reloj t_2 y t_5, pero $z = 0$ durante t_2 y $z = 1$ durante t_5. De modo similar, $w = 1$ durante t_1 y t_8, pero $z = 0$ durante t_1 y $z = 1$ durante t_8. Esto significa que z no está determinada sólo por el valor presente en w, así que debe haber diferentes estados en el circuito que determinen el valor de z.

8.1.1 DIAGRAMA DE ESTADO

El primer paso en el diseño de una máquina de estado finito consiste en determinar cuántos estados se necesitan y cuáles transiciones son posibles de un estado a otro. No hay un procedimiento determinado para esta tarea. El diseñador debe pensar detenidamente en lo que la máquina debe hacer. Una buena forma de comenzar es elegir un estado en particular como estado *inicial*; éste es el estado en el que debe entrar el circuito cuando se encienda por primera vez o cuando se le aplique una señal *reset* (reinicio). Para nuestro ejemplo, supongamos que el estado inicial se llama estado A. Siempre que la entrada w sea 0, el circuito no necesita hacer nada y, por tanto, cada flanco activo del reloj debe dar como resultado que el circuito permanezca en el estado A. Cuando w se vuelve igual a 1, la máquina debe reconocerlo y pasar a un estado distinto, al cual llamaremos estado B. Esta transición se lleva a cabo en el siguiente flanco activo del reloj después que w se ha vuelto igual a 1. En el estado B, igual que en el A, el circuito debe mantener el valor de la salida z en 0, ya que aún no ha visto $w = 1$ para dos ciclos del reloj consecutivos. Cuando se halla en el estado B, si w es 0 en el siguiente flanco activo del reloj el circuito debe regresar al estado A. Sin embargo, si $w = 1$ cuando se encuentra en el estado B el circuito debe cambiar

Ciclo del reloj	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figura 8.2 Secuencias de señales de entrada y salida.

a un tercer estado, llamado *C*, y luego debe generar una salida $z = 1$. El circuito ha de permanecer en el estado *C* siempre que $w = 1$ y debe seguir manteniendo $z = 1$. Cuando *w* se vuelve 0, la máquina tiene que moverse de regreso al estado *A*. Como la descripción anterior maneja todos los valores posibles de la entrada *w* que la máquina puede encontrar en sus distintos estados, podemos concluir que se necesitan tres estados para implementar la máquina deseada.

Ahora que hemos determinado de manera informal las posibles transiciones entre los estados, describiremos un procedimiento más formal para diseñar el circuito secuencial correspondiente. El comportamiento de un circuito secuencial puede describirse de varias formas. El método conceptualmente más simple es utilizar una representación gráfica en forma de un *diagrama de estado*, que es una gráfica que representa los estados del circuito como nodos (círculos) y las transiciones entre estados como arcos con dirección. El diagrama de estado de la figura 8.3 define el comportamiento que corresponde a nuestra especificación. Los estados *A*, *B* y *C* aparecen como nodos en el diagrama. El nodo *A* representa el estado inicial y también el estado en el que entrará el circuito *después* que se aplique una entrada $w = 0$. En este estado la salida *z* debe ser 0, lo cual se indica como $A/z=0$ en el nodo. El circuito debe permanecer en el estado A siempre que $w = 0$, lo que se indica por medio de un arco con una etiqueta $w = 0$ que se origina y termina en este nodo. La primera ocurrencia de $w = 1$ (después de la condición $w = 0$) se registra al pasar del estado *A* al estado *B*. Esta transición se indica en la gráfica por medio de un arco que se origina en *A* y termina en *B*. La etiqueta $w = 1$ en este arco denota el valor de entrada que ocasiona la transición. En el estado *B* la salida permanece en 0, lo cual se indica como $B/z=0$ en el nodo.

Cuando el circuito se halla en el estado *B*, cambiará al estado *C* si *w* aún es igual a 1 en el siguiente flanco activo del reloj. En el estado *C* la salida *z* se vuelve igual a 1. Si *w* permanece en 1 durante los ciclos subsiguientes del reloj, el circuito permanecerá en el estado *C* manteniendo $z = 1$. No obstante, si *w* se vuelve 0 cuando el circuito está ya sea en el estado *B* o en el estado *C*, el siguiente flanco activo del reloj producirá una transición al estado *A*.

En el diagrama indicamos que la entrada *Reset* se utiliza para forzar el circuito a entrar en el estado *A*, lo cual es posible independientemente del estado en el que se encuentre el circuito.

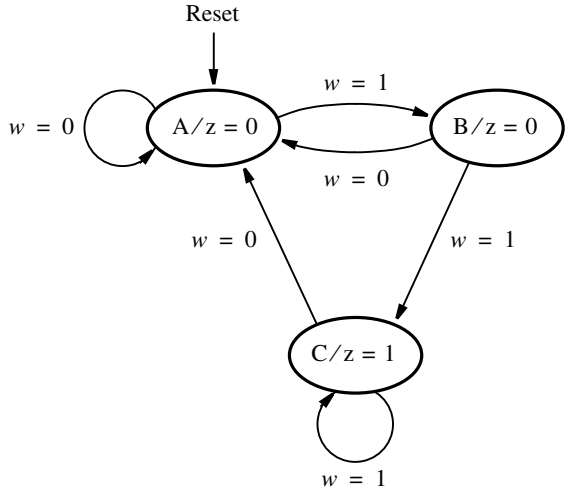


Figura 8.3 Diagrama de estado de un circuito secuencial simple.

Podríamos tratar *Reset* simplemente como otra entrada al circuito y mostrar una transición de cada estado al estado inicial *A* bajo el control de la entrada *Reset*. Esto complicaría el diagrama de modo innecesario. Los estados de una máquina de estado finito se implementan usando flip-flops. Como por lo general éstos tienen la capacidad reset, como vimos en el capítulo 7, podemos suponer que la entrada *Reset* se utiliza para establecer todos en 0 mediante tal capacidad. Indicaremos esto como se muestra en la figura 8.3 para mantener los diagramas tan simples como sea posible.

8.1.2 TABLA DE ESTADO

Aun cuando el diagrama de estado brinda una descripción del comportamiento de un circuito secuencial que es fácil de comprender, para proseguir con la implementación del circuito es conveniente traducir la información contenida en el diagrama de estado en forma de tabla. En la figura 8.4 se muestra la *tabla de estado* para nuestro circuito secuencial. Ahí se indican todas las transiciones de cada *estado presente* al *estado siguiente* para los diferentes valores de la señal de entrada. Nótese que la salida z se especifica respecto al estado presente, en concreto, el estado en el que se halla el circuito en el momento presente. Obsérvese también que no incluimos la entrada *Reset*; en vez de ello hicimos la suposición implícita de que el primer estado en la tabla es el estado inicial.

Ahora mostramos los pasos de diseño que producirán el circuito final. Para explicar los conceptos de diseño básicos, primero seguimos un procedimiento tradicional de realizar manualmente cada paso de diseño. Luego se presenta una explicación de las técnicas de diseño automatizadas que usan herramientas de diseño asistido por computadora (CAD).

8.1.3 ASIGNACIÓN DE ESTADOS

La tabla de estado de la figura 8.4 define los tres estados en términos de las letras *A*, *B* y *C*. Cuando se implementa en un circuito lógico, cada estado se representa por medio de una valoración (combinación de valores) en particular de las *variables de estado*. Cada variable de estado puede implementarse en forma de un flip-flop. Como deben producirse tres estados, basta utilizar dos variables de estado. Sean estas variables y_1 y y_2.

Ahora podemos adaptar el diagrama de bloque general de la figura 8.1 a nuestro ejemplo, como se muestra en la figura 8.5, para indicar la estructura del circuito que implementa la máquina de estado finito buscada. Dos flip-flops representan las variables de estado. En la figura no hemos especificado el tipo de flip-flops que se utilizarán; este tema se aborda en la siguiente

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Figura 8.4 Tabla de estado para el circuito secuencial de la figura 8.3.

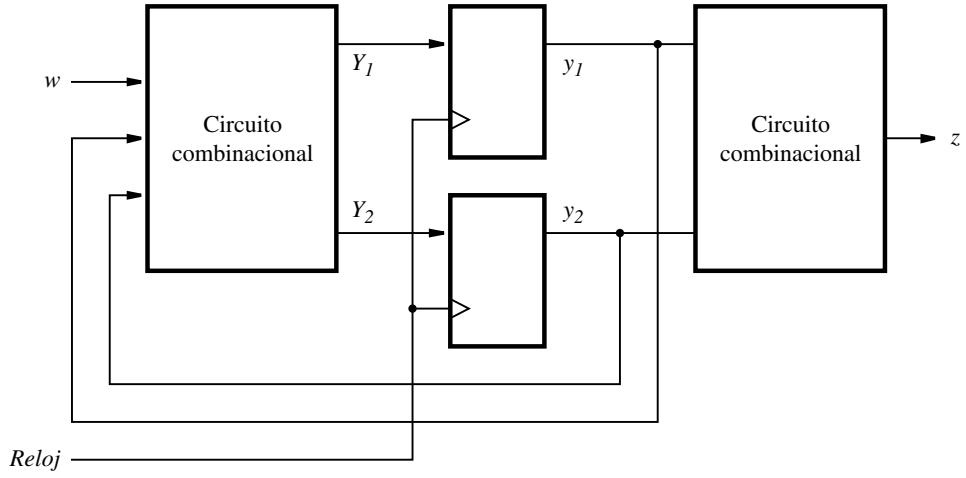


Figura 8.5 Un circuito secuencial general con la entrada w, la salida z y dos flip-flops de estado.

subsección. A partir de la especificación de las figuras 8.3 y 8.4, la salida z está determinada sólo por el estado presente del circuito. Por tanto, el diagrama de bloque de la figura 8.5 muestra que z es una función sólo de y_1 y y_2; nuestro diseño es más del tipo Moore. Necesitamos diseñar un circuito combinacional que utilice y_1 y y_2 como señales de entrada y que genere una señal de salida correcta z para todas las combinaciones posibles de estas entradas.

Las señales y_1 y y_2 también se alimentan de regreso en el circuito combinacional que determina el siguiente estado de la FSM. Este circuito también utiliza la señal de entrada principal w. Sus salidas son las dos señales, Y_1 y Y_2, que se emplean para establecer el estado de los flip-flops. Cada flanco activo del reloj hará que los flip-flops cambien su estado a los valores de Y_1 y Y_2 en ese momento. Por consiguiente, Y_1 y Y_2 se llaman *variables del estado siguiente* y y_1 y y_2 *variables del estado presente*. Debemos diseñar un circuito combinacional con entradas w, y_1 y y_2 tales que, para todas las combinaciones de estas entradas, las salidas Y_1 y Y_2 ocasionarán que la máquina pase al estado siguiente que satisface la especificación. El paso siguiente en el proceso de diseño consiste en crear una tabla de verdad que defina este circuito, así como el circuito que genera z.

Para producir la tabla de verdad deseada, asignamos una combinación específica de las variables y_1 y y_2 para cada estado. Una asignación posible se da en la figura 8.6, donde los estados A, B y C se representan por medio de $y_2y_1 = 00$, 01 y 10, respectivamente. La cuarta combinación, $y_2y_1 = 11$, no se necesita en este caso.

El tipo de tabla dado en la figura 8.6 suele llamarse *tabla de asignación de estados*. Esta tabla puede servir directamente como una tabla de verdad para la salida z con las entradas y_1 y y_2. Aun cuando para las funciones del estado siguiente Y_1 y Y_2 la tabla no posee la apariencia de una tabla de verdad normal, porque hay dos columnas separadas en la tabla para cada valor de w, es evidente que la tabla incluye toda la información que define las funciones del estado siguiente en términos de las combinaciones de las entradas w, y_1 y y_2.

	Estado presente	Estado siguiente		Salida
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

Figura 8.6 Tabla de asignación de estados para el circuito secuencial de la figura 8.4.

8.1.4 ELECCIÓN DE FLIP-FLOPS Y DERIVACIÓN DE LAS EXPRESIONES DE ESTADO SIGUIENTE Y DE SALIDA

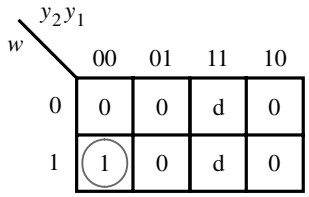
A partir de la tabla de asignación de estados de la figura 8.6 podemos derivar las expresiones lógicas de las funciones del estado siguiente y de la salida. Pero primero tenemos que decidir el tipo de flip-flops que se utilizarán en el circuito. La opción más sencilla es usar flip-flops tipo D, pues en este caso los valores de Y_1 y Y_2 simplemente se registran en los flip-flops para que se vuelvan los nuevos valores de y_1 y y_2. En otras palabras, si las entradas a los flip-flops se llaman D_1 y D_2, entonces estas señales son las mismas que Y_1 y Y_2. Nótese que el diagrama de la figura 8.5 corresponde exactamente a este uso de los flip-flops tipo D. Para otros tipos de flip-flops, como el JK, la relación entre la variable del estado siguiente y las entradas a un flip-flop no es tan sencilla; consideraremos esta situación en la sección 8.7.

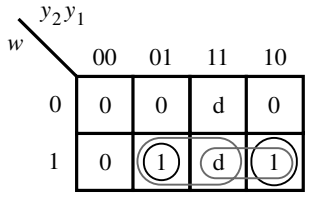
Las expresiones lógicas requeridas pueden deducirse como se muestra en la figura 8.7. Usamos mapas de Karnaugh para facilitar al lector la verificación de la validez de las expresiones. Recuérdese que en la figura 8.6 sólo necesitamos tres de las cuatro combinaciones binarias posibles para representar los estados. La cuarta combinación, $y_2y_1 = 11$, nunca debe ocurrir en el circuito porque está restringido a cambiar sólo a los estados A, B y C; en consecuencia, tal vez optemos por tratar esta combinación como una condición no-importa. Los cuadrados no-importa resultantes en los mapas de Karnaugh se indican por medio de la letra d. Al usar condiciones no-importa para simplificar las expresiones obtenemos

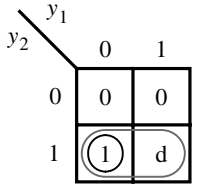
$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = w(y_1 + y_2)$$
$$z = y_2$$

Si no utilizamos las condiciones no-importa, entonces las expresiones resultantes son ligeramente más complejas; se muestran en el área sombreada de la figura 8.7.

Puesto que $D_1 = Y_1$ y $D_2 = Y_2$, el circuito lógico correspondiente a las expresiones anteriores se implementa como se muestra en la figura 8.8. Obsérvese que se incluye una señal de reloj y que el circuito se proporciona con una capacidad reset activa en nivel bajo. Conectar la entrada clear de los flip-flops a una señal *Resetn* externa, como se muestra en la figura, brinda una forma







	Ignora los no-importa	Utiliza los no-importa
Y_1	$w\bar{y}_1\bar{y}_2$	$w\bar{y}_1\bar{y}_2$
Y_2	$wy_1\bar{y}_2 + w\bar{y}_1y_2$	$wy_1 + wy_2 = w(y_1 + y_2)$
z	$\bar{y}_1y_2$	y_2

Figura 8.7 Derivación de las expresiones lógicas para el circuito secuencial de la figura 8.6.

sencilla de forzar a que el circuito entre en un estado conocido. Si aplicamos la señal *Resetn* = 0 al circuito, entonces los dos flip-flops se borrarán, poniendo la FSM en el estado $y_2y_1 = 00$.

8.1.5 DIAGRAMA DE TIEMPO

Para comprender cabalmente la operación del circuito de la figura 8.8 consideremos su diagrama de tiempo presentado en la figura 8.9. En el diagrama se describen las formas de onda de las señales que corresponden a las secuencias de valores de la figura 8.2.

Como estamos utilizando flip-flops disparados por el flanco positivo, todos los cambios en las señales ocurren poco tiempo después del flanco positivo del reloj. La cantidad de demora del flanco del reloj depende de los retrasos de propagación a través de los flip-flops. Obsérvese que la señal de entrada *w* también se muestra para cambiar ligeramente después del flanco activo del reloj. Esta suposición es buena porque en un sistema digital típico una entrada como *w* sería sólo una salida de otro circuito sincronizado por el mismo reloj. La sincronización de las señales de entrada con la señal de reloj se estudia en la sección 10.3.

Un aspecto clave que ha de notarse es que aun cuando *w* cambia poco tiempo después del flanco activo del reloj y, por tanto, su valor es igual a 1 (o a 0) para casi todo el ciclo de reloj, no ocurrirá ningún cambio en el circuito hasta el comienzo del siguiente ciclo de reloj cuando el

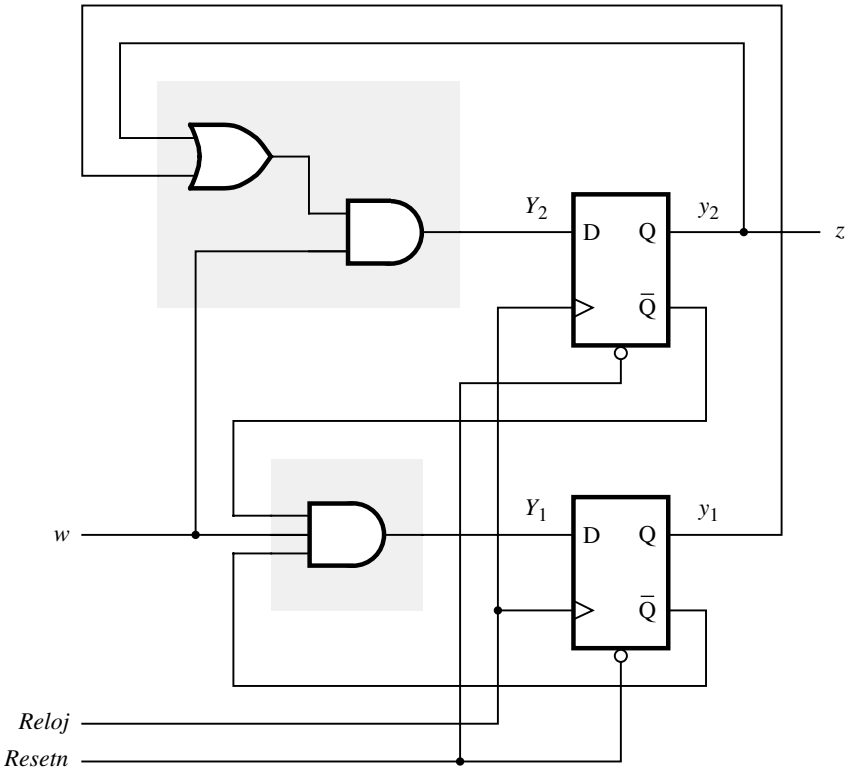


Figura 8.8 Implementación final del circuito secuencial de la figura 8.7.

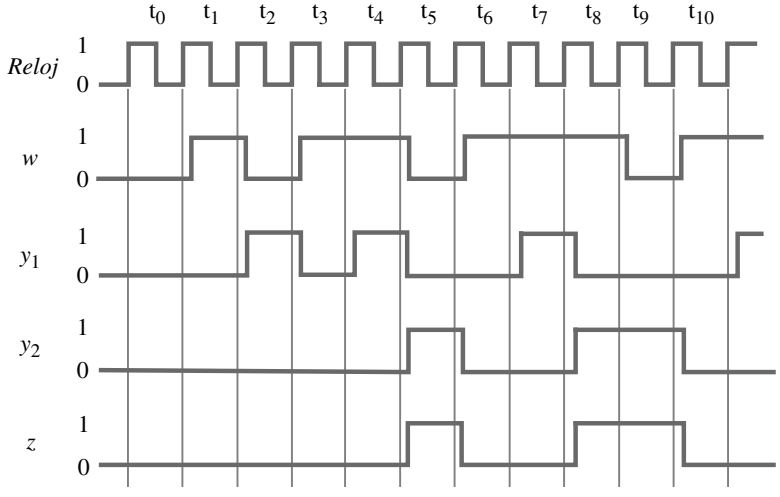


Figura 8.9 Diagrama de tiempo para el circuito de la figura 8.8.

flanco positivo haga que los flip-flops cambien de estado. Por ende, el valor de w debe ser igual a 1 para dos ciclos de reloj si el circuito va a alcanzar el estado C y a generar la salida $z = 1$.

8.1.6 RESUMEN DE LOS PASOS DE DISEÑO

Podemos resumir los pasos que comprende el diseño de un circuito síncrono secuencial como sigue:

1. Se obtiene la especificación del circuito buscado.
2. Se derivan los estados para la máquina seleccionando primero un estado inicial. Luego, con la especificación del circuito se consideran todas las combinaciones de las entradas al circuito y se crean estados nuevos según se requiera para que la máquina responda a estas entradas. Para seguir la pista de los estados a medida que se visitan, se crea un diagrama de estado. Cuando está completo, el diagrama de estado muestra todos los estados en la máquina y proporciona las condiciones en las que el circuito pasa de un estado a otro.
3. Se elabora una tabla de estado a partir del diagrama de estado. De manera opcional, tal vez sea conveniente crear directamente la tabla de estado en el paso 2, en vez de crear primero un diagrama de estado.
4. En nuestro circuito secuencial de ejemplo sólo había tres estados; por consiguiente, la creación de una tabla de estado que no tuviera más estados que los necesarios era simple. Sin embargo, en la práctica es común lidiar con circuitos con muchos estados. En estos casos es poco probable que el primer intento por derivar una tabla de estado produzca resultados óptimos. Es casi seguro que tendremos más estados que los realmente necesarios. Esto puede corregirse mediante un procedimiento que reduce al mínimo el número de estados. Estudiaremos el proceso de minimización de estados en la sección 8.6.
5. Se decide el número de variables de estado necesarias para representar todos los estados y realizar la asignación de éstos. Hay muchas asignaciones de estados posibles para un circuito secuencial. Algunas son mejores que otras. En el ejemplo anterior usamos lo que parecía una asignación de estados natural. Retomaremos este ejemplo en la sección 8.2 y mostraremos que una asignación diferente puede conducir a un circuito más simple.
6. Se elige el tipo de flip-flops que van a usarse en el circuito. Se derivan las expresiones lógicas del estado siguiente para controlar las entradas a todos los flip-flops y luego se derivan las expresiones lógicas para las salidas del circuito. Hasta ahora hemos empleado sólo flip-flops D. Consideraremos otros tipos de flip-flops en la sección 8.7.
7. Se implementa el circuito según lo indiquen las expresiones lógicas.

Ejemplo 8.1 Hemos ilustrado los pasos de diseño utilizando un circuito secuencial muy simple. Desde el punto de vista del lector, un circuito que detecta que una señal de entrada estuvo en alto para dos pulsos de reloj consecutivos tal vez no tenga mucho significado práctico. Ahora consideraremos un ejemplo muy relacionado con la aplicación práctica.

En la sección 7.14 presentamos el concepto de bus y mostramos las conexiones que tuvieron que hacerse para permitir que el contenido de un registro se transfiriera a otro registro. En el

circuito de la figura 7.55 se muestra cómo usar los buffers triestado para colocar el contenido de un registro seleccionado en el bus y cómo los datos de éste pueden cargarse en un registro. En la figura 7.57 se indica cómo un mecanismo de control que intercambie el contenido de los registros $R1$ y $R2$ puede realizarse con un registro de corrimiento. Ahora diseñaremos el mecanismo de control deseado aplicando el enfoque de máquina de estado finito.

El contenido de los registros $R1$ y $R2$ puede intercambiarse utilizando el registro $R3$ como un lugar de almacenamiento temporal como sigue: el contenido de $R2$ se carga primero en $R3$ usando las señales de control $R2_{out} = 1$ y $R3_{in} = 1$. Luego el contenido de $R1$ se transfiere a $R2$ usando $R1_{out} = 1$ y $R2_{in} = 1$. Finalmente, el contenido de $R3$ (que es el contenido anterior de $R2$) se transfiere a $R1$ usando $R3_{out} = 1$ y $R1_{in} = 1$. Como este paso completa el intercambio requerido, para indicar que la tarea está terminanda estableceremos la señal $Done = 1$. Suponga que el intercambio se realiza en respuesta a un pulso en una señal de entrada llamada w, que perdura un ciclo de reloj. En la figura 8.10 se muestran las señales externas que el circuito de control deseado comprende. En la figura 8.11 se presenta un diagrama de estado para un circuito secuencial que genera las señales de control de salida en la secuencia requerida. Observe que para mantener el diagrama simple hemos indicado las señales de salida únicamente cuando son iguales a 1. En todos los demás casos las señales de salida son iguales a 0.

En el estado inicial, A, no se indica ninguna transferencia y todas las señales de salida son 0. El circuito permanece en este estado hasta que llega una solicitud para intercambio en la forma de w que cambia a 1. En el estado B las señales necesarias para transferir el contenido de $R2$ a $R3$ se validan. El siguiente flanco activo del reloj coloca estos contenidos en $R3$. Esto también ocasiona que el circuito cambie al estado C, independientemente de si w es igual a 0 o a 1. En este estado las señales para transferir $R1$ a $R2$ se validan. La transferencia se realiza en el siguiente flanco activo del reloj y el circuito cambia al estado D con independencia del valor de w. La transferencia final, de $R3$ a $R1$, se lleva a cabo en el flanco de reloj que deja el estado D, el cual también causa que el circuito regrese al estado A.

En la figura 8.12 se presenta la misma información en una tabla de estado. Como hay cuatro estados, es preciso usar dos variables de estado, y_2 y y_1. Una asignación de estados sencilla donde a los estados A, B, C y D se les asignan las combinaciones $y_2y_1 = 00$, 01, 10 y 11, respectivamente, conduce a la tabla de asignación de estados de la figura 8.13. Al emplear esta asignación

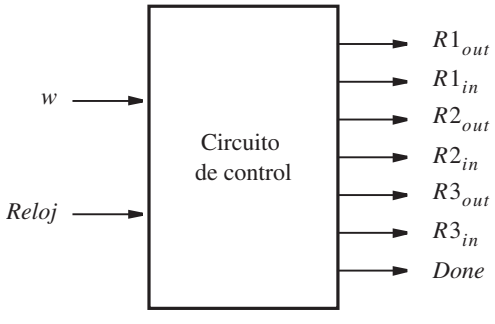


Figura 8.10 Señales que se necesitan en el ejemplo 8.1.

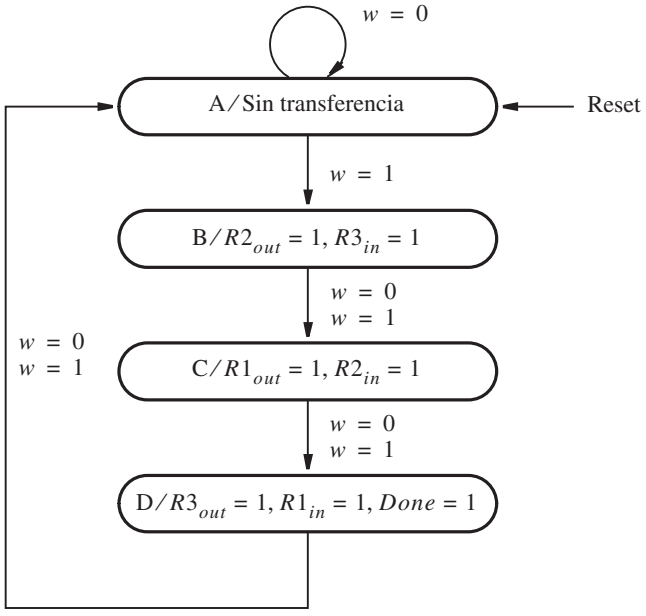


Figura 8.11 Diagrama de estado para el ejemplo 8.1.

Estado presente	Estado siguiente		Salidas						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Figura 8.12 Tabla de estado para el ejemplo 8.1.

	Estado presente	Estado siguiente		Salidas						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 0	1 0	0	0	1	0	0	1	0
C	1 0	1 1	1 1	1	0	0	1	0	0	0
D	1 1	0 0	0 0	0	1	0	0	1	0	1

Figura 8.13 Tabla de asignación de estados para el circuito secuencial de la figura 8.12.

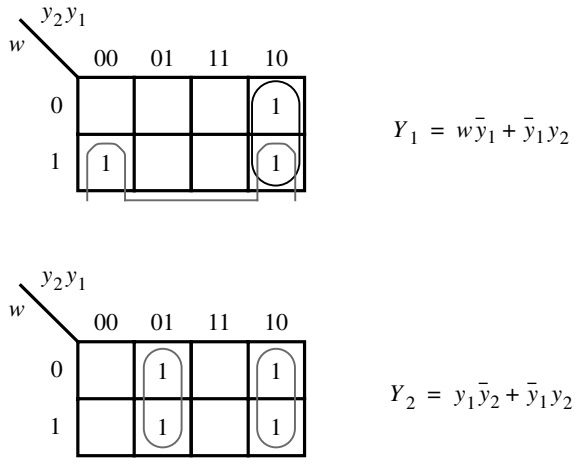


Figura 8.14 Derivación de las expresiones del estado siguiente para el circuito secuencial de la figura 8.13.

y flip-flops D, las expresiones del estado siguiente pueden derivarse como se muestra en la figura 8.14. Éstas son

$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

Las señales de control de salida se derivan como

$$R1_{out} = R2_{in} = \bar{y}_1 y_2$$
$$R1_{in} = R3_{out} = Done = y_1 y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

Estas expresiones conducen al circuito de la figura 8.15. Este circuito parece más complejo que el registro de corrimiento de la figura 7.57, pero sólo tiene dos flip-flops, en vez de tres.

8.2 EL PROBLEMA DE LA ASIGNACIÓN DE ESTADOS

Una vez presentados los conceptos básicos relativos al diseño de los circuitos secuenciales debemos repasar algunos detalles de las diferentes alternativas. En la sección 8.1.6 sugerimos que algunas asignaciones de estado pueden ser mejores que otras. Para ilustrarlo podemos reconsiderar el ejemplo de la figura 8.4. Ya sabemos que la asignación de estados de la figura 8.6 conduce a un circuito de apariencia simple de la figura 8.8. Pero, ¿la FSM de la figura 8.4 puede implementarse con un circuito aún más simple utilizando una asignación de estados diferente?

En la figura 8.16 se proporciona una alternativa. En este caso representamos los estados A, B, C con las combinaciones $y_2y_1 = 00, 01$ y 11, respectivamente. La combinación restante, $y_2y_1 = 10$, no es necesaria y la trataremos como una condición no-importa. Si elegimos de nuevo imple-

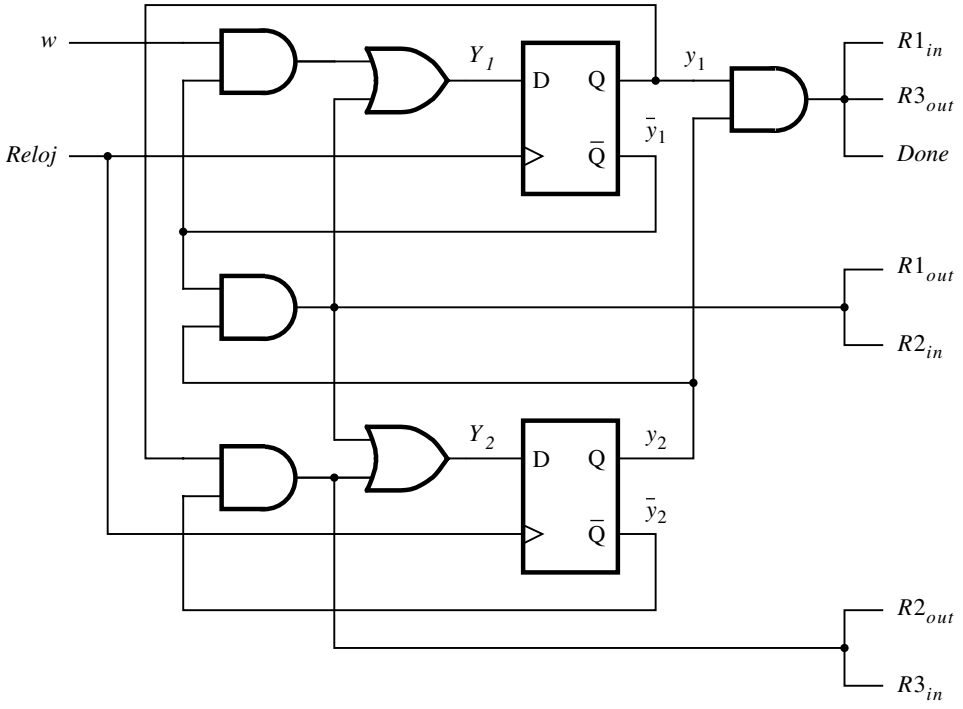


Figura 8.15 Implementación final del circuito secuencial de la figura 8.13.

	Estado presente	Estado siguiente		Salida
		$w = 0$	$w = 1$	z
	y_2y_1	Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

Figura 8.16 Asignación de estados mejorada para el circuito secuencial de la figura 8.4.

mentar el circuito usando flip-flops D, las expresiones del estado siguiente y de la salida deducidas de la figura serán

$$Y_1 = D_1 = w$$
$$Y_2 = D_2 = wy_1$$
$$z = y_2$$

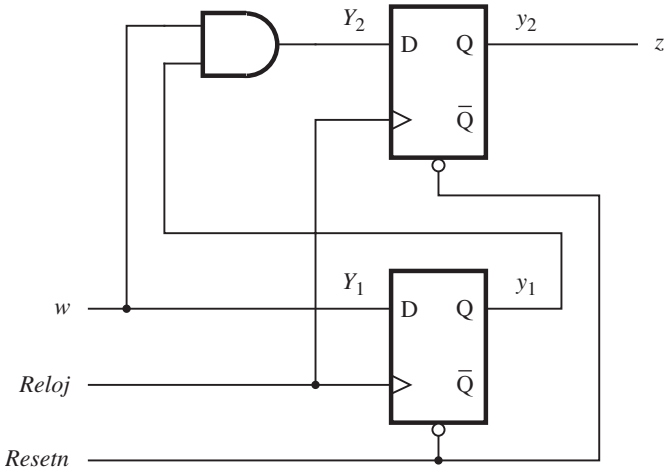


Figura 8.17 Circuito final para la asignación de estados mejorada de la figura 8.16.

Estas expresiones definen el circuito mostrado en la figura 8.17. Al comparar este circuito con el de la figura 8.8 se observa que el costo del circuito nuevo es menor porque necesita menos compuertas.

En general, los circuitos son mucho más grandes que nuestro ejemplo, y las distintas asignaciones de estado pueden tener un efecto sustancial en el costo de la implementación final. Si bien es muy deseable, a menudo resulta imposible encontrar la mejor asignación de estados para un circuito grande. El enfoque exhaustivo de intentar todas las asignaciones de estado posibles no es práctico porque el número de asignaciones es enorme. Las herramientas CAD suelen realizar la asignación de estados usando técnicas heurísticas, que casi siempre están patentadas y sus detalles se publican pocas veces.

Ejemplo 8.2

En la figura 8.13 utilizamos una asignación de estados sencilla para el circuito secuencial de la figura 8.12. Considere ahora el efecto de intercambiar las combinaciones asignadas a los estados *C* y *D*, como se muestra en la figura 8.18. Luego, las expresiones del estado siguiente son

$$Y_1 = w\overline{y}_2 + y_1\overline{y}_2$$
$$Y_2 = y_1$$

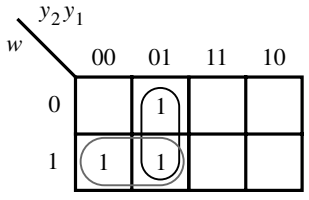
según se derivó en la figura 8.19. Las expresiones de salida son

$$R1_{out} = R2_{in} = y_1y_2$$
$$R1_{in} = R3_{out} = Done = \overline{y}_1y_2$$
$$R2_{out} = R3_{in} = y_1\overline{y}_2$$

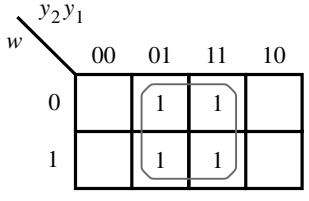
Estas expresiones conducen a un circuito ligeramente más simple que el de la figura 8.15.

	Estado presente	Estado siguiente		Salidas						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	*Done*
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 1	1 1	0	0	1	0	0	1	0
C	1 1	1 0	1 0	1	0	0	1	0	0	0
D	1 0	0 0	0 0	0	1	0	0	1	0	1

Figura 8.18 Asignación de estados mejorada para el circuito secuencial de la figura 8.12.



$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$



$$Y_2 = y_1$$

Figura 8.19 Derivación de las expresiones del estado siguiente para el circuito secuencial de la figura 8.18.

8.2.1 CODIFICACIÓN DE 1 ACTIVO

Otra posibilidad interesante es utilizar tantas variables de estado como estados haya en un circuito secuencial. De acuerdo con este método, para cada estado todas las variables de estado excepto una son iguales a 0. La variable cuyo valor es 1 se considera "activa" (*hot*). El método se conoce como *codificación de 1 activo (one-hot encoding)*.

En la figura 8.20 se muestra cómo una asignación de estados de 1 activo puede aplicarse al circuito secuencial de la figura 8.4. Como hay tres estados, es preciso utilizar tres variables de estado. La asignación elegida representará los estados A, B y C usando las combinaciones $y_3y_2y_1 =$ 001, 010 y 100, respectivamente. Las cinco combinaciones restantes de las variables de estado no

	Estado presente	Estado siguiente		Salida
		$w = 0$	$w = 1$	
	$y_3 y_2 y_1$	$Y_3 Y_2 Y_1$	$Y_3 Y_2 Y_1$	z
A	0 0 1	0 0 1	0 1 0	0
B	0 1 0	0 0 1	1 0 0	0
C	1 0 0	0 0 1	1 0 0	1

Figura 8.20 Asignación de estados de 1 activo para el circuito secuencial de la figura 8.4.

se utilizan, por lo que pueden tratarse como condiciones no-importa en la derivación de las expresiones del estado siguiente y de la salida. Al usar esta asignación las expresiones resultantes son

$$Y_1 = \overline{w}$$
$$Y_2 = w y_1$$
$$Y_3 = w \overline{y}_1$$
$$z = y_3$$

Estas expresiones no son más simples que las obtenidas utilizando la asignación de estado de la figura 8.16. Aun cuando en este caso la asignación de 1 activo no es favorable, este enfoque resulta atractivo en muchos casos.

Ejemplo 8.3

La asignación de estados de 1 activo puede aplicarse al circuito secuencial de la figura 8.12 como se indica en la figura 8.21. Se necesitan cuatro variables de estado, y los estados A, B, C y D se codifican como $y_4 y_3 y_2 y_1 = 0001$, 0010, 0100 y 1000, respectivamente. Al tratar las 12 combina-

	Estado presente	Estado siguiente		Salidas						
		$w = 0$	$w = 1$							
	$y_4 y_3 y_2 y_1$	$Y_4 Y_3 Y_2 Y_1$	$Y_4 Y_3 Y_2 Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 0 0 1	0 0 0 1	0 0 1 0	0	0	0	0	0	0	0
B	0 0 1 0	0 1 0 0	0 1 0 0	0	0	1	0	0	1	0
C	0 1 0 0	1 0 0 0	1 0 0 0	1	0	0	1	0	0	0
D	1 0 0 0	0 0 0 1	0 0 0 1	0	1	0	0	1	0	1

Figura 8.21 Asignación de estados de 1 activo para el circuito secuencial de la figura 8.12.

ciones restantes de las variables de estado como condiciones no-importa las expresiones del estado siguiente son

$$Y_1 = \overline{w}y_1 + y_4$$
$$Y_2 = wy_1$$
$$Y_3 = y_2$$
$$Y_4 = y_3$$

Es importante observar que podemos derivar estas expresiones por simple inspección del diagrama de estado de la figura 8.11. El flip-flop y_1 debe establecerse en 1 si la FSM está en el estado A y $w = 0$, o si la FSM está en el estado D; por consiguiente, $Y_1 = \overline{w}y_1 + y_4$. El flip-flop y_2 debe establecerse en 1 si el estado presente es A y $w = 1$; por tanto, $Y_2 = wy_1$. Los flip-flops y_3 y y_4 deben establecerse en 1 si la FSM está en el estado B o C, respectivamente; por ende, $Y_3 = y_2$ y $Y_4 = y_3$.

Las expresiones de salida son justo las salidas de los flip-flops, tales que

$$R1_{out} = R2_{in} = y_3$$
$$R1_{in} = R3_{out} = Done = y_4$$
$$R2_{out} = R3_{in} = y_2$$

Estas expresiones son más simples que las derivadas en el ejemplo 8.2, pero se requieren cuatro flip-flops, en vez de dos.

Una característica relevante de la asignación de estados de 1 activo es que a menudo conduce a expresiones de salida más simples que las asignaciones con el número mínimo de variables de estado. Las expresiones de salida más simples pueden generar en un circuito más rápido. Por ejemplo, si las salidas del circuito secuencial son simplemente las salidas de los flip-flops, como ocurre en nuestro ejemplo, entonces estas señales de salida son válidas en cuanto los flip-flops cambian su estado. Si hay expresiones de salida más complejas, entonces debe tomarse en cuenta el retraso de propagación por las compuertas que implementan estas expresiones. Consideraremos esta situación en la sección 8.8.2.

Los ejemplos expuestos hasta ahora muestran que hay muchas formas de implementar una máquina de estado finito como un circuito secuencial. Es probable que cada implementación tenga un costo diferente y características de sincronización distintas. En la sección siguiente presentamos otra formar de modelar las FSM que conduce a más posibilidades.

8.3 MODELO DE ESTADO TIPO MEALY

Nuestros ejemplos introductorios fueron circuitos secuenciales en los que cada estado tenía valores específicos de las señales de salida asociadas con él. Como explicamos al principio del capítulo, se dice que estas máquinas de estado finito corresponden al tipo Moore. Ahora exploraremos el concepto de máquinas tipo Mealy en las que los valores de salida se generan con base tanto en el estado del circuito como en los valores presentes en las entradas. Esto brinda mayor flexibilidad en el diseño de circuitos secuenciales. Presentaremos las máquinas tipo Mealy usando una versión un tanto modificada de un ejemplo anterior.

La esencia del primer circuito secuencial en la sección 8.1 es generar la salida $z = 1$ siempre que una segunda ocurrencia de la entrada $w = 1$ se detecte en ciclos del reloj consecutivos. La especificación indica que la salida z debe ser igual a 1 en el ciclo del reloj que sigue a la

detección de la segunda ocurrencia de $w = 1$. Supóngase ahora que eliminamos este requisito y especificamos ahora que la salida z debe ser igual a 1 en el mismo ciclo del reloj cuando se detecta la segunda ocurrencia de $w = 1$. Por tanto, una secuencia de entrada-salida adecuada puede ser como se muestra en la figura 8.22. Para ver cómo podemos producir el comportamiento dado en esta tabla comenzamos seleccionando un estado inicial, A. Siempre que $w = 0$, la máquina debe permanecer en el estado A, produciendo la salida $z = 0$. Cuando $w = 1$, la máquina debe pasar a un estado nuevo, B, para registrar el hecho de que ha ocurrido una entrada igual a 1. Si w aún es igual a 1 cuando la máquina está en el estado B, lo cual ocurre si $w = 1$ durante al menos dos ciclos consecutivos de reloj, la máquina debe permanecer en el estado B y producir una salida $z = 1$. En cuanto w se vuelve 0, z debe volverse 0 de inmediato y la máquina debe regresar al estado A en el siguiente flanco activo del reloj. Por ende, el comportamiento especificado en la figura 8.22 puede lograrse con una máquina de dos estados, la cual tiene un diagrama de estado como el que aparece en la figura 8.23. Sólo se requieren dos estados porque hemos permitido que el valor de salida dependa del valor presente en la entrada así como del estado presente de la máquina. El diagrama indica que si la máquina se halla en el estado A, seguirá en él si $w = 0$ y la salida será 0. Esto se señala mediante un arco con la etiqueta $w = 0/z = 0$. Cuando w se vuelve 1, la salida permanece en 0 hasta que la máquina pasa al estado B en el siguiente flanco activo del reloj. Esto se denota por medio de un arco de A a B con la etiqueta $w = 1/z = 0$. En el estado B la salida será 1 si $w = 1$, y la máquina permanecerá en el estado B, como se indica con la etiqueta $w = 1/z = 1$ en el arco correspondiente. Sin embargo, si $w = 0$ en el estado B, entonces la salida será 0 y una transición al estado A ocurrirá en el siguiente flanco activo del reloj. Un aspecto importante que debe comprenderse es que durante el ciclo de reloj actual el valor de salida corresponde a la etiqueta en el arco que proviene del nodo del estado presente.

Podemos implementar la FSM de la figura 8.23 usando los mismos pasos de diseño que en la sección 8.1. La tabla de estado se muestra en la figura 8.24. En ella se indica que la salida z depende del valor presente en la entrada w y no sólo del estado presente. La figura 8.25 propor-

Ciclo del reloj	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

Figura 8.22 Secuencias de las señales de entrada y salida.

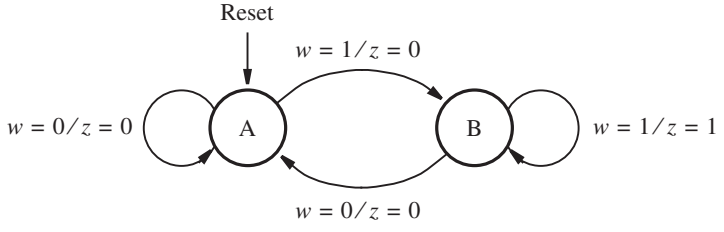


Figura 8.23 Diagrama de estado de una FSM que realiza la tarea de la figura 8.22.

Estado presente	Estado siguiente		Salida z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

Figura 8.24 Tabla de estado para la FSM de la figura 8.23.

	Estado presente	Estado siguiente		Salida	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Figura 8.25 Tabla de asignación de estados para la FSM de la figura 8.24.

ciona la tabla de asignación de estados. Como sólo hay dos estados, es suficiente utilizar una sola variable de estado, y. Si suponemos que y se produce con un flip-flop D, las expresiones requeridas de estado siguiente y de salida son

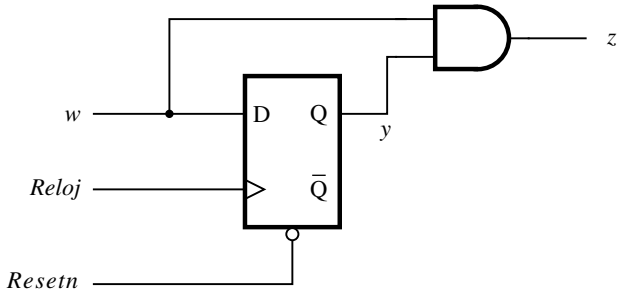
$$Y = D = w$$

$$z = wy$$

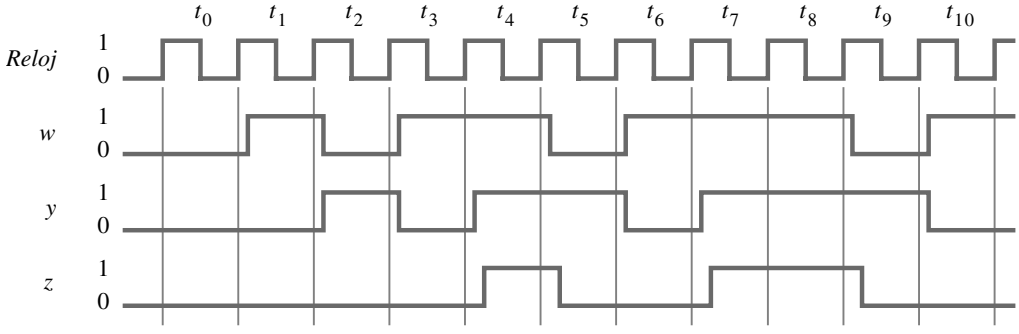
El circuito resultante se presenta en la figura 8.26 junto con un diagrama de tiempo. Este último corresponde a las secuencias de entrada-salida de la figura 8.22.

La mayor flexibilidad de las FSM tipo Mealy a menudo desemboca en la realización de un circuito más simple. Desde luego éste parece ser el caso en nuestros ejemplos que produjeron los circuitos de las figuras 8.8, 8.17 y 8.26, suponiendo que el requisito de diseño sólo es detectar dos ocurrencias consecutivas de la entrada w que son iguales a 1. Debemos observar, no obstante, que el circuito de la figura 8.26 no es el mismo en términos del comportamiento de salida que los circuitos de las figuras 8.8 y 8.17. La diferencia es un corrimiento de un ciclo de reloj en la señal de salida de la figura 8.26*b*. Si queremos producir exactamente el mismo comportamiento de salida utilizando el modelo Mealy, podríamos modificar el circuito de la figura 8.26*a* añadiendo otro flip-flop como se muestra en la figura 8.27. Este flip-flop simplemente retarda la señal de salida, Z, un ciclo de reloj respecto a z, como se indica en el diagrama de tiempo. Al hacer este cambio, el circuito tipo Mealy se convierte efectivamente en un circuito tipo Moore con la salida Z. Nótese que el circuito de la figura 8.27 es en esencia el mismo que el circuito de la figura 8.17.

Ejemplo 8.4 En el ejemplo 8.1 consideramos el circuito de control necesario para intercambiar el contenido de dos registros, implementado como una máquina de estado finita tipo Moore. La misma tarea



a) Circuito



b) Diagrama de tiempo

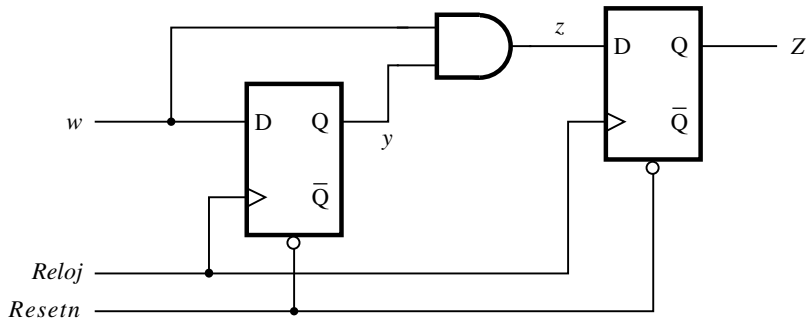
Figura 8.26 Implementación de la máquina de estado finito de la figura 8.25.

puede lograrse con una FSM tipo Mealy, como se indica en la figura 8.28. El estado A aún sirve como el estado reset. Pero en cuanto w cambia de 0 a 1, las señales de control de salida $R2_{out}$ y $R3_{in}$ se validan, y son válidas hasta el principio del ciclo de reloj siguiente, cuando el circuito dejará el estado A y pasará a B. En el estado B las salidas $R1_{out}$ y $R2_{in}$ se validan tanto para $w = 0$ como $w = 1$. Finalmente, en el estado C el intercambio se completa al validar $R3_{out}$ y $R1_{in}$.

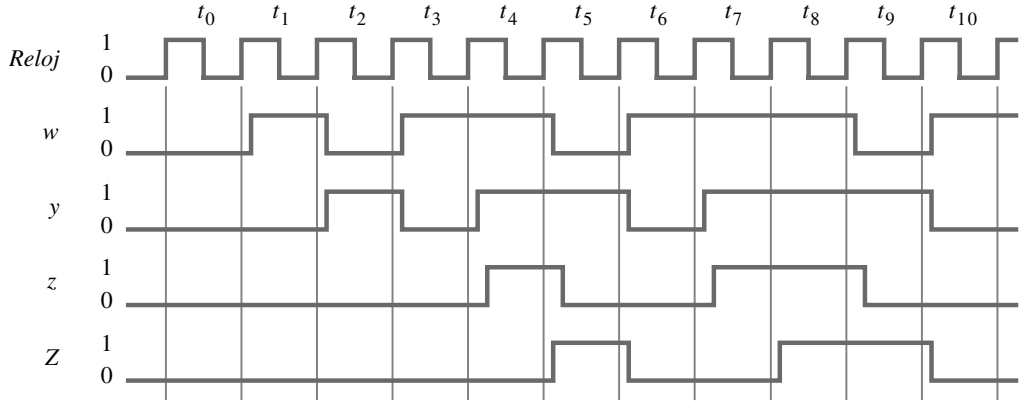
La realización de circuito de control tipo Mealy requiere tres estados, lo que no necesariamente implica un circuito más simple porque aún se necesitan los dos flip-flops para implementar las variables de estado. La diferencia más importante en comparación con la realización del tipo Moore es la sincronización de las señales de salida. Un circuito que implementa la FSM de la figura 8.28 genera las señales de control de salida un ciclo de reloj antes que los circuitos derivados en los ejemplos 8.1 y 8.2.

Obsérvese también que al utilizar la FSM de la figura 8.28, todo el proceso de intercambio del contenido de $R1$ y $R2$ demora tres ciclos de reloj, empezando y terminando en el estado A. Al emplear la FSM tipo Moore en el ejemplo 8.1, el proceso de intercambio abarca cuatro ciclos de reloj antes que el circuito vuelva al estado A.

Supóngase que deseamos implementar esta FSM utilizando la codificación de 1 activo. Por tanto, se precisan tres flip-flops, y es posible asignar las combinaciones $y_3y_2y_1 = 001$, 010 y 100 a los estados A, B y C, respectivamente. Al examinar el diagrama de estado de la figura 8.28,



a) Circuito



b) Diagrama de tiempo

Figura 8.27 Circuito que implementa la especificación de la figura 8.2.

podemos derivar las ecuaciones del estado siguiente por inspección. La entrada al flip-flop y_1 debe tener el valor 1 si la FSM se halla en el estado A y $w = 0$, o si la FSM está en el estado C; por consiguiente, $Y_1 = \overline{w}y_1 + y_3$. El flip-flop y_2 debe establecerse en 1 si la FSM se encuentra en el estado A y $w = 1$; por ende, $Y_2 = wy_1$. El flip-flop y_3 debe establecerse en 1 si el estado presente es B; por tanto, $Y_3 = y_2$. La derivación de las expresiones de salida, la cual dejamos como ejercicio para el lector, también puede realizarse por inspección. El circuito correspondiente se muestra en la figura 7.58, en la sección 7.14, donde se dedujo aplicando un enfoque diferente.

La exposición anterior aborda los principios básicos relativos al diseño de los circuitos secuenciales. Aunque resulta esencial comprender estos principios, el enfoque manual utilizado en los ejemplos es difícil y tedioso cuando se trata de circuitos más grades. Ahora mostraremos cómo se usan las herramientas CAD para simplificar en buena medida la tarea de diseño.

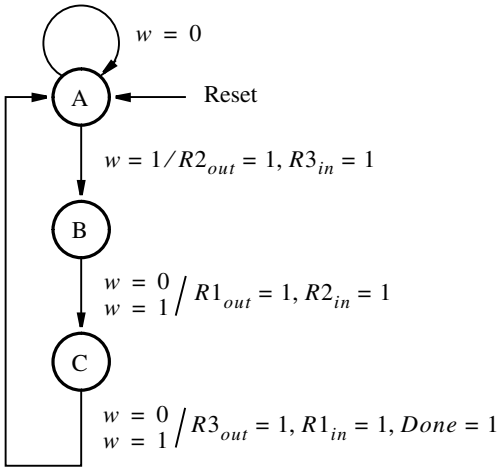


Figura 8.28 Diagrama de estado para el ejemplo 8.4.

8.4 DISEÑO DE MÁQUINAS DE ESTADO FINITO CON HERRAMIENTAS CAD

Se dispone de herramientas CAD muy modernas para el diseño de máquinas de estado finito y las presentamos en esta sección. Una forma rudimentaria de emplear tales herramientas para el diseño de una FSM podría ser como sigue: el diseñador usa las técnicas manuales descritas antes para derivar un circuito con flip-flops y compuertas lógicas a partir de un diagrama de estado. Este circuito se introduce en el sistema CAD trazando un diagrama esquemático o escribiendo código estructural en lenguaje de descripción de hardware (HDL, *hardware description language*). El diseñador utiliza entonces el sistema CAD para simular el comportamiento del circuito y las herramientas CAD para implementar en forma automática el circuito en un chip, por ejemplo en un PLD.

Resulta tedioso sintetizar manualmente un circuito a partir de un diagrama de estado. Como las herramientas CAD se diseñaron para evitar esa tarea, se han desarrollado formas más atractivas de utilizarlas para el diseño de FSM. Un mejor enfoque consiste en introducir directamente el diagrama de estado en el sistema CAD y realizar todo el proceso de síntesis de forma automática. Las herramientas CAD soportan este método en dos formas principales. Otro enfoque radica en permitir al diseñador trazar el diagrama de estado utilizando una herramienta gráfica parecida a la herramienta de captura esquemática. El diseñador dibuja círculos para representar los estados y arcos para representar las transiciones de un estado a otro, e indica las salidas que la máquina debe generar. El enfoque más popular consiste en utilizar HDL para escribir código que represente el diagrama de estado, como se describe enseguida.

Muchos HDL proveen constructores que permiten al diseñador representar un diagrama de estado. Para mostrar cómo se hace esto, proporcionaremos código de VHDL que representa la máquina simple diseñada manualmente como el primer ejemplo de la sección 8.1. Luego utilizaremos las herramientas CAD para sintetizar un circuito que implementa la máquina en un chip.

8.4.1 CÓDIGO DE VHDL PARA FSM TIPO MOORE

VHDL no define una manera estándar de describir una máquina de estado finito. Por tanto, cuanto nos apeguemos a la sintaxis de VDHL requerida, habrá más de una forma de describir una FSM. En la figura 8.29 se presenta un ejemplo de código de VHDL para la FSM de la figura 8.3. En aras de facilitar la explicación, las líneas de código están numeradas en el lado izquierdo. En las líneas 1 a 6 se declara una entidad llamada *simple*, que tiene puertos de entrada *Clock*, *Resetn* y *w*, y un puerto de salida, *z*. En la línea 7 se usa el nombre *Behavior* para el cuerpo de arquitectura, pero desde luego en su lugar podría utilizarse cualquier nombre válido en VHDL.

En la línea 8 se introduce la palabra reservada TYPE, que es una función de VHDL que no hemos ocupado hasta ahora. Nos permite crear un tipo de señal definido por el usuario. El nuevo tipo de señal se llama *State_type*, y el código especifica que una señal de este tipo puede tener tres valores posibles: *A*, *B* o *C*. En la línea 9 se define una señal llamada *y* que es del tipo *State_type*. La señal *y* se usa en el cuerpo de arquitectura para representar las salidas de los flip-flops que implementan los estados de la FSM. El código no especifica el número de bits representado por *y*. En vez de ello, indica que *y* puede tener los tres valores simbólicos *A*, *B* y *C*. Esto significa que no hemos especificado el número de flip-flops de estado que deben utilizarse para la FSM. Como veremos enseguida, el compilador de VHDL elige de forma automática un número apropiado de flip-flops de estado cuando se sintetiza un circuito para implementar la máquina. También elige la asignación de estado para los estados *A*, *B* y *C*. Algunos sistemas CAD, como Quartus II, suponen que el primer estado enumerado en la instrucción TYPE (línea 8) es el estado reset para la máquina. La asignación de estado que tiene todas las salidas de flip-flop iguales a 0 se utiliza para este estado. Más adelante en esta sección mostraremos cómo es posible especificar manualmente la codificación de estados en el código de VHDL si así se desea.

Una vez definida una señal para representar los flip-flops de estado, el siguiente paso consiste en especificar las transiciones entre estados. En la figura 8.29 se presenta una forma de describir el diagrama de estado; esto se representa por medio del proceso de las líneas 11 a 37. La instrucción PROCESS describe la máquina de estado finito como un circuito secuencial. Se basa en el enfoque que aplicamos para describir un flip-flop D disparado por flanco en la sección 7.12.2. Las señales utilizadas por el proceso son *Clock*, *Resetn*, *w* y *y*, y la única señal modificada por el proceso es *y*. Las señales de entrada que pueden hacer que el proceso cambie la señal *y* son *Clock* y *Resetn*; por consiguiente, aparecen en la lista de sensibilidad. Nótese que *w* no está incluida en esa lista, ya que un cambio en su valor no puede afectar a *y* hasta que ocurre un cambio en la señal *Clock*.

En las líneas 13 y 14 se especifica que la máquina debe entrar en el estado *A*, el estado reset, si *Resetn* = 0. Puesto que la condición para la instrucción IF no depende de la señal de reloj, el estado reset es asíncrono, razón por la que *Resetn* se incluye en la lista de sensibilidad en la línea 11.

Cuando la señal reset no se valida, la instrucción ELSIF en la línea 15 especifica que el circuito espera el flanco positivo de la señal de reloj (*Clock*). Obsérvese que la condición ELSIF es la misma que la condición usada para describir un flip-flop D disparado por flanco positivo de la figura 7.39. El comportamiento de *y* se define por medio de la instrucción CASE de las líneas 16 a 35. Corresponde al diagrama de estado de la figura 8.3. Como la instrucción CASE está dentro de la instrucción ELSIF, cualquier cambio en *y* sólo puede ocurrir como resultado de un flanco positivo del reloj. En otras palabras, la condición ELSIF implica que *y* debe implementarse como la salida de uno o más flip-flops. Cada cláusula WHEN en la instrucción CASE representa un estado de la máquina. Por ejemplo, la cláusula WHEN en las líneas 17 a 22 describe el compor-

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;

3  ENTITY simple IS
4      PORT ( Clock, Resetn, w     : IN     STD_LOGIC ;
5             z                    : OUT  STD_LOGIC ) ;
6  END simple ;

7  ARCHITECTURE Behavior OF simple IS
8      TYPE State_type IS (A, B, C) ;
9      SIGNAL y : State_type ;
10 BEGIN
11     PROCESS ( Resetn, Clock )
12     BEGIN
13         IF Resetn = '0' THEN
14             y <= A ;
15         ELSIF (Clock'EVENT AND Clock = '1') THEN
16             CASE y IS
17                 WHEN A =>
18                     IF w = '0' THEN
19                         y <= A ;
20                     ELSE
21                         y <= B ;
22                     END IF ;
23                 WHEN B =>
24                     IF w = '0' THEN
25                         y <= A ;
26                     ELSE
27                         y <= C ;
28                     END IF ;
29                 WHEN C =>
30                     IF w = '0' THEN
31                         y <= A ;
32                     ELSE
33                         y <= C ;
34                     END IF ;
35             END CASE ;
36         END IF ;
37     END PROCESS ;
38     z <= '1' WHEN y = C ELSE '0' ;
39 END Behavior ;
```

Figura 8.29 Código de VHDL para la FSM de la figura 8.3.

tamiento de la máquina cuando se halla en el estado *A*. Según el principio de la instrucción IF en la línea 18, cuando la FSM se encuentra en el estado *A*, si $w = 0$ la máquina debe permanecer en el estado *A*; pero si $w = 1$, debe cambiar al estado *B*. Las cláusulas WHEN en la instrucción CASE corresponden exactamente al diagrama de estado de la figura 8.3.

La parte final de la descripción de la máquina de estado aparece en la línea 38. Especifica que si la máquina se encuentra en el estado *C*, entonces la salida *z* debe ser 1; de lo contrario, *z* debe ser 0.

8.4.2 SÍNTESIS DEL CÓDIGO DE VHDL

Para dar un ejemplo del circuito producido por una herramienta de síntesis, sintetizamos el código de la figura 8.29 a fin de implementarlo en un CPLD. La síntesis dio como resultado dos flip-flops, con entradas Y_1 y Y_2 y salidas y_1 y y_2. Las expresiones del estado siguiente generadas por la herramienta de síntesis son

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = wy_1 + wy_2$$

La expresión de salida es

$$z = y_2$$

Estas expresiones corresponden al caso de la figura 8.7 cuando el estado no utilizado $y_2y_1 = 11$ se trata como condición no-importa en los mapas de Karnaugh para Y_1, Y_2 y z.

En la figura 8.30 se representa una parte del circuito FSM implementado en un CPLD. Para mantener esta figura simple sólo se muestran los recursos lógicos utilizado por las dos macroceldas que implementan y_1, y_2 y z. Las partes de las macroceldas utilizadas para el circuito se destacan en gris.

La entrada *w* al circuito aparece conectada a uno de los cables de interconexión del CPLD. El nodo de origen en el chip que genera *w* no se muestra. Podría ser un pin de entrada o que *w* fuera la salida de otra macrocelda, suponiendo que el CPLD pueda contener otro sistema de circuitos conectado a nuestra FSM. La señal *Clock* se asigna a un pin en el chip que se dedica para que lo usen las señales del reloj. Desde este pin dedicado, un *cable global* distribuye la señal de reloj a todos los flip-flops del chip. El cable global distribuye la señal de reloj a los flip-flops de modo que la diferencia en el tiempo de llegada, o *desviación de reloj*, de la señal de reloj en cada flip-flop se minimice. El concepto de desviación de reloj se estudia en la sección 10.3. Un cable global también se usa para la señal reset.

La macrocelda superior de la figura 8.30 produce la variable de estado y_1. La otra macrocelda genera y_2. Para la señal y_1, la macrocelda superior produce el término producto requerido, como se muestra. Los otros cables del término producto requerido en la macrocelda no aparecen en la figura, pero cada uno se establece en 0, de modo que la compuerta OR no se afecte. La salida de la compuerta OR pasa por la compuerta XOR, cuya otra entrada es 0. Aunque la compuerta XOR no afecta el comportamiento de este circuito, excepto para causar un pequeño retraso de propagación, es una parte de la macrocelda y no puede evitarse cuando se implemente nuestro circuito. La salida de la compuerta XOR maneja el flip-flop que representa a y_1. La entrada select del multiplexor se establece en 1, de modo que la señal y_1 pasa por el buffer triestado. De igual manera que la compuerta XOR, este buffer no se necesita en nuestro circuito, pero como está presente en la macrocelda debe utilizarse; por consiguiente, su señal de control enable para

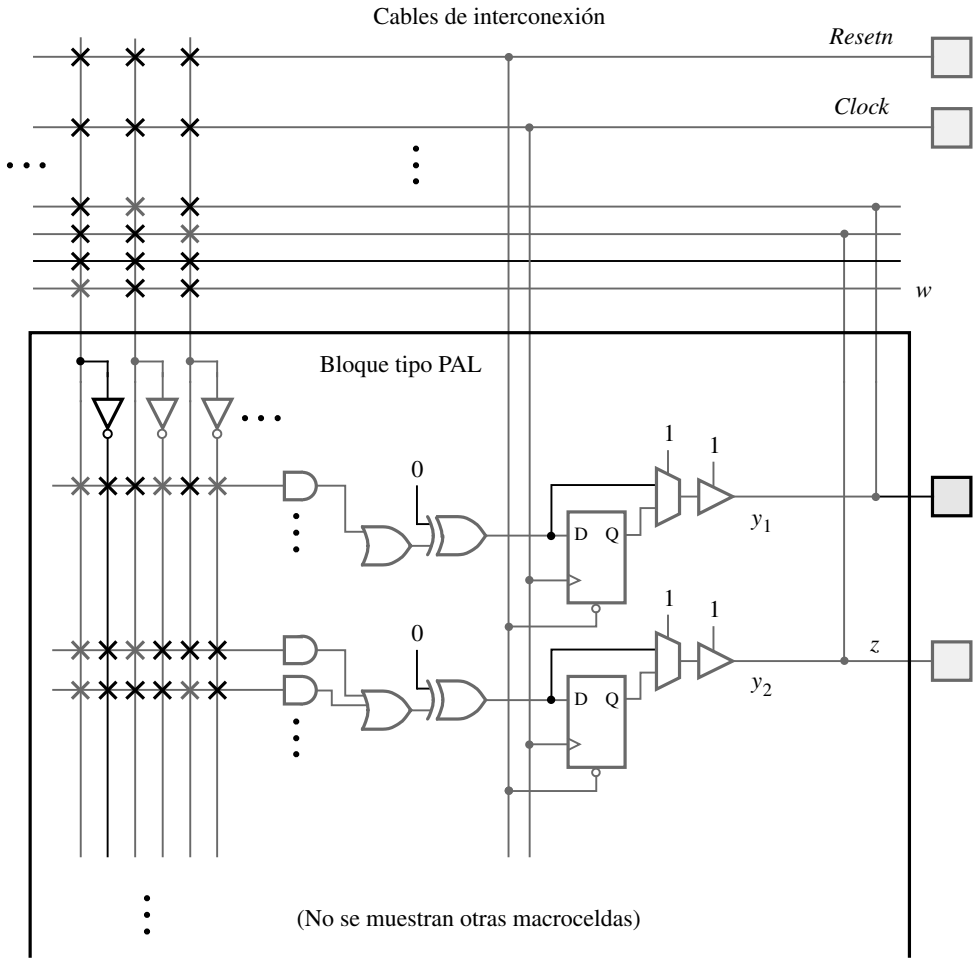


Figura 8.30 Implementación de la FSM de la figura 8.3 en un CPLD.

habilitar la salida se establece en 1. La señal y_1 se conecta a los cables de interconexión en el CPLD y se retroalimenta a las macroceldas. Obsérvese que aun cuando y_1 no es una salida del circuito, utiliza una trayectoria de señal que está conectada a uno de los pines del chip. Por consiguiente, este pin no puede utilizarse para ningún otro propósito. La implementación de y_2 es similar a la de y_1, excepto porque están involucrados los dos términos producto. La señal y_2 está conectada al pin etiquetado z, que produce la señal de salida requerida.

En la figura 8.31 se ilustra cómo podría asignarse el circuito a los pines de un CPLD pequeño en un paquete PLCC de 44 pines. En la figura se muestra el corte transversal de la parte superior del chip, lo que revela una vista conceptual de las dos macroceldas de la figura 8.30, las cuales se indican en gris. Nuestro circuito simple utiliza sólo una pequeña parte del dispositivo.

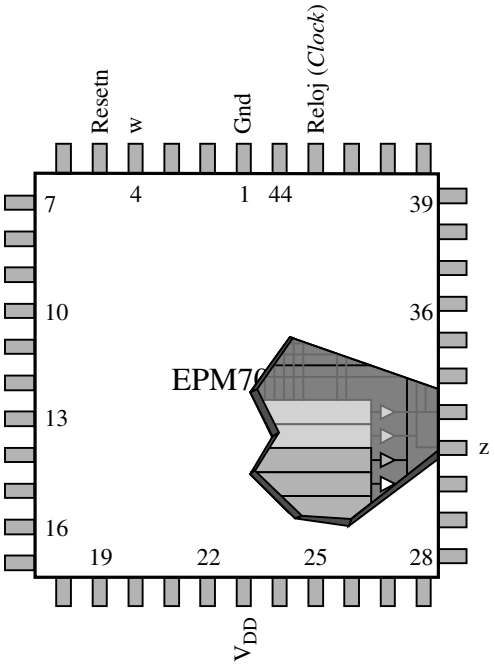


Figura 8.31 El circuito de la figura 8.30 en un CPLD pequeño.

8.4.3 SIMULACIÓN Y PRUEBA DEL CIRCUITO

El comportamiento del circuito implementado en el chip CPLD puede probarse utilizando la simulación de tiempo, como se describe en la figura 8.32. Ahí se proporcionan las formas de onda correspondientes al diagrama de tiempo de la figura 8.9, suponiendo que se usa un periodo de reloj de 100 ns. La señal *Resetn* se establece en 0 al principio de la simulación y luego en 1. El circuito produce la salida $z = 1$ para un ciclo de reloj después que w ha sido igual a 1 durante dos ciclos sucesivos de reloj. Cuando w es 1 durante tres ciclos de reloj, z se vuelve 1 para dos ciclos de reloj, como debería ser. Mostramos los cambios en el estado utilizando las letras A, B y C para facilitar la lectura. (El simulador incluido en el libro en realidad muestra los códigos binarios correspondientes a los estados.)

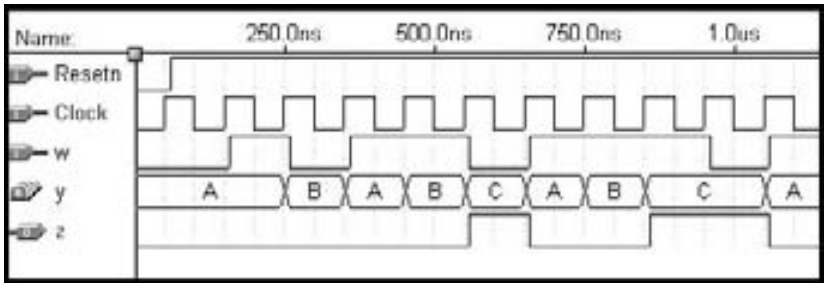


Figura 8.32 Resultados de la simulación para el circuito de la figura 8.30.

Luego de examinar la salida de la simulación, debemos preguntarnos si es posible concluir que el circuito funciona correctamente y satisface todos los requisitos. Para nuestro circuito simple no es difícil responder porque el circuito sólo tiene una entrada y su comportamiento es sencillo. Resulta fácil ver que el circuito funciona bien. Sin embargo, en general es difícil asegurar con cierta confianza si un circuito secuencial funcionará de manera adecuada para todas las secuencias de entrada posibles, ya que un gran número de patrones de entrada puede ser posible. Para las máquinas de estado finito grandes el diseñador debe pensar detenidamente en los patrones de entrada que pueden emplearse en la simulación a fin de realizar pruebas.

8.4.4 UN ESTILO ALTERNO DE CÓDIGO DE VHDL

Mencionamos antes en esta sección que VHDL no especifica una manera estándar de escribir código que represente una máquina de estado finito. El código de la figura 8.29 es sólo una posibilidad. Un segundo ejemplo de código para nuestra máquina simple se presenta en la figura 8.33. Sólo se muestra el cuerpo de arquitectura porque la declaración de entidad es la misma que la de la figura 8.29. Se utilizan dos señales para representar el estado de la máquina. La señal llamada *y_present* corresponde al estado presente y *y_next* al estado siguiente. En términos de la notación utilizada en la sección 8.1.3, *y_present* es lo mismo que *y*, y *y_next* es *Y*. No podemos emplear *y* para denotar el estado presente y *Y* para el estado siguiente en el código porque VHDL no distingue entre letras minúsculas y mayúsculas. Las dos señales, *y_present* y *y_next*, son del tipo *State_type*.

La máquina está especificada por dos procesos separados. El primero describe la tabla de estado como un circuito combinacional. Utiliza una instrucción CASE para dar el valor de *y_next* por cada valor de *y_present* y *w*. El código puede relacionarse con la forma general de las FSM de la figura 8.5. El proceso corresponde al circuito combinacional del lado izquierdo de la figura.

El segundo proceso introduce los flip-flops en el circuito. Estipula que después de cada flanco positivo del reloj la señal *y_present* debe tomar el valor de la señal *y_next*. El proceso también especifica que *y_present* debe tomar el valor de *A* cuando $Resetn = 0$, lo que proporciona el reset asíncrono.

Hemos mostrado dos estilos de código de VHDL para nuestro ejemplo de FSM. Es probable que el circuito producido por el compilador de VHDL para cada versión del código sea un tanto distinto ya que, como el lector bien sabe en este punto, hay muchas formas de implementar una función lógica. Sin embargo, los circuitos producidos a partir de las dos versiones del código brindan exactamente la misma funcionalidad.

8.4.5 RESUMEN DE LOS PASOS DE DISEÑO CUANDO SE USAN HERRAMIENTAS CAD

En la sección 8.1.6 resumimos los pasos de diseño necesarios para derivar muchos circuitos secuenciales. Ahora hemos visto que las herramientas CAD pueden realizar gran parte del trabajo de manera automática. Sin embargo, es importante darse cuenta que las herramientas CAD no han remplazado *todos* los pasos manuales. Respecto a la lista dada en la sección 8.1.6, los dos primeros pasos, en los que se obtiene la especificación de la máquina y se deriva un diagrama de estado, aún deben hacerse manualmente. Al dar la información del diagrama de estado como entrada, las herramientas CAD entonces pueden realizar en forma automática las tareas necesarias

```
ARCHITECTURE Behavior OF simple IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS (Clock, Resetn)
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figura 8.33 Estilo alterno de código de VHDL para la FSM de la figura 8.3.

para generar un circuito con compuertas lógicas y flip-flops. Además de los pasos de diseño presentado en la sección 8.1.6, debemos añadir la etapa de pruebas y simulación. Diferimos un análisis detallado de este tema para el capítulo 11.

8.4.6 ESPECIFICACIÓN DE LA ASIGNACIÓN DE ESTADOS EN EL CÓDIGO DE VHDL

En la sección 8.2 vimos que la asignación de estados puede afectar la complejidad del circuito diseñado. Un objetivo obvio del proceso de asignación de estados es minimizar el costo de la implementación. La función de costo que debe optimarse puede ser simplemente el número de compuertas y flip-flops. Pero también podría basarse en otras consideraciones representativas de la estructura de los chips PLD utilizados para implementar el diseño. Por ejemplo, el software CAD puede tratar de encontrar codificaciones de estado que minimicen el número total de términos AND necesarios en el circuito resultante cuando el chip objetivo sea un CPLD.

En el código de VHDL es posible especificar la asignación de estado que debe emplearse, mas no hay una manera estandarizada de hacerlo. Por consiguiente, aun cuando se apega a la sintaxis de VHDL, cada sistema CAD permite un método ligeramente distinto de especificar la asignación de estados. El sistema Quartus II recomienda que la asignación de estados se realice utilizando la función de atributo de VHDL. Un *atributo* se refiere a algún tipo de información respecto a un objeto en el código de VHDL. A todas las señales se les asigna automáticamente un número de atributos *predefinidos* asociados. Un ejemplo es el atributo EVENT que usamos para especificar un flanco del reloj, como en Clock'EVENT.

Además de los atributos predefinidos, es posible crear un atributo definido por el usuario. El atributo *definido por el usuario* puede usarse para asociar algún tipo de información deseada con un objeto en el código de VHDL. En Quartus II la asignación de estados manual puede efectuarse con un atributo definido por el usuario asociado con el tipo *State_type*. Esto se ilustra en la figura 8.34, la cual muestra las primeras líneas de la arquitectura de la figura 8.33 con la adición de un atributo definido por el usuario. Primero definimos el nuevo atributo, llamado ENUM_ENCODING, que tiene el tipo STRING. La siguiente línea asocia ENUM_ENCODING con el tipo *State_type* y especifica que el atributo tiene el valor "00 01 11". Cuando se traduce el código de VHDL, el compilador de Quartus II utiliza el valor de ENUM_ENCODING para hacer la asignación de estados $A = 00$, $B = 01$ y $C = 11$.

El atributo ENUM_ENCODING es específico de Quartus II. Por tanto, tal vez no podamos utilizar este método de asignación de estado en otros sistemas CAD. Una manera diferente de dar la asignación de estados, para que funcione con cualquier sistema CAD, se muestra en la

```
ARCHITECTURE Behavior OF simple IS
    TYPE State_TYPE IS (A, B, C) ;
    ATTRIBUTE ENUM_ENCODING                 : STRING ;
    ATTRIBUTE ENUM_ENCODING OF State_type : TYPE IS "00 01 11" ;
    SIGNAL y_present, y_next                : State_type ;
BEGIN
    .
    .
    .
```

Figura 8.34 Un atributo definido por el usuario para la asignación manual de estados.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY simple IS
    PORT ( Clock, Resetn, w  : IN   STD_LOGIC ;
           z                 : OUT  STD_LOGIC ) ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
    CONSTANT A : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
    CONSTANT B : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
    CONSTANT C : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN OTHERS =>
                y_next <= A ;
        END CASE ;
    END PROCESS ;
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figura 8.35 Uso de constantes para la asignación manual de estados.

figura 8.35. En vez de utilizar el tipo *State_type* como en los ejemplos anteriores, *y_present* y *y_next* se definen como señales STD_LOGIC_VECTOR de dos bits. Cada uno de los nombres simbólicos para los tres estados, *A*, *B* y *C*, están definidos como constantes, correspondiendo el valor de cada constante a la codificación deseada. Obsérvese que la sintaxis para la asignación de un valor a una constante utiliza el operador de asignación := en vez del operador <= empleado para las señales. Cuando el código se traduce, el compilador de VHDL remplaza los nombres simbólicos *A*, *B* y *C* con los valores asignados a sus constantes.

La instrucción CASE que define el diagrama de estado es idéntica a la de la figura 8.33 con una excepción. VHDL requiere que la instrucción CASE para *y_present* incluya una cláusula WHEN para todos los valores posibles de *y_present*. En la figura 8.33 *y_present* sólo puede poseer los tres valores *A*, *B* y *C* porque tiene el tipo de estado *State_type*. Pero como *y_present* es una señal STD_LOGIC_VECTOR en la figura 8.35, debemos proporcionar una cláusula WHEN OTHERS, como se muestra. En la práctica, la máquina nunca debe entrar en el estado no utilizado, el cual corresponde a *y_present* = 10. Como dijimos antes, hay una remota posibilidad de que esto ocurra a causa de un comportamiento erróneo del circuito. Como opción pragmática, hemos especificado que la FSM debe regresar al estado reset si ocurre un error como éste.

8.4.7 ESPECIFICACIÓN DE FSM TIPO MEALY CON VHDL

Una FSM tipo Mealy puede especificarse de una manera similar a una FSM tipo Moore. En la figura 8.36 se proporciona el código de VHDL completo para la FSM de la figura 8.23. Las transiciones de estado se describen de la misma forma que en nuestro ejemplo de VHDL original de la figura 8.29. La señal *y* representa los flip-flops de estado y *State_type* especifica que *y* puede tener los valores *A* y *B*. Comparado con el código de la figura 8.29, la principal diferencia en el caso de una FSM tipo Mealy es el modo en que se escribe el código para la salida. En la figura 8.36 la salida *z* se define con una instrucción CASE. Establece que cuando la FSM se halla en el estado *A*, *z* debe ser 0, pero cuando se encuentra en el estado *B*, *z* debe tomar el valor de *w*. Esta instrucción CASE describe de manera apropiada la lógica necesaria para *z*, pero tal vez no sea evidente por qué se usa una segunda instrucción CASE en el código en vez de especificar el valor de *z* dentro de la instrucción CASE que define las transiciones de estado. La razón es que la instrucción CASE para las transiciones de estado está anidada dentro de la instrucción IF que espera que ocurra un flanco del reloj. Por consiguiente, si colocamos el código para *z* dentro de esta instrucción CASE, entonces el valor de *z* sólo podría cambiar como resultado de un flanco del reloj. Esto no satisface los requisitos de la FSM tipo Mealy, pues el valor de *z* debe depender no sólo del estado de la máquina sino también de la entrada *w*.

La implementación de la FSM especificada en la figura 8.36 en un chip CPLD produce las mismas ecuaciones que derivamos en forma manual en la sección 8.3. Los resultados de la simulación para el circuito sintetizado aparecen en la figura 8.37. La forma de onda de entrada para *w* es la misma que la que utilizamos para la máquina tipo Moore de la figura 8.32. Nuestra máquina tipo Mealy se comporta correctamente, con *z* volviéndose 1 justo después del comienzo del segundo ciclo consecutivo de reloj en el que *w* es 1.

En los resultados de la simulación que hemos dado en esta sección, todos los cambios en la entrada *w* ocurren inmediatamente después de un flanco positivo del reloj. Esto se basa en la suposición planteada en la sección 8.1.5 de que en un circuito real *w* estaría sincronizada respecto al reloj que controla a la FSM. En la figura 8.38 ilustramos un problema que puede surgir si *w* no cumple esta especificación. En este caso hemos supuesto que los cambios en *w* ocurren en el

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn, w  : IN   STD_LOGIC ;
           z                 : OUT  STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 8.36 Código de VHDL para la máquina Mealy de la figura 8.23.

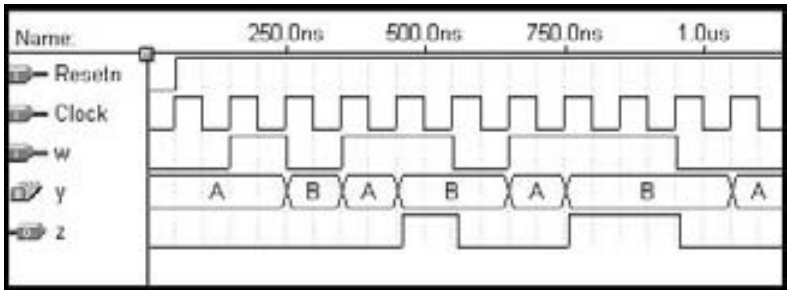


Figura 8.37 Resultados de la simulación para la máquina Mealy.

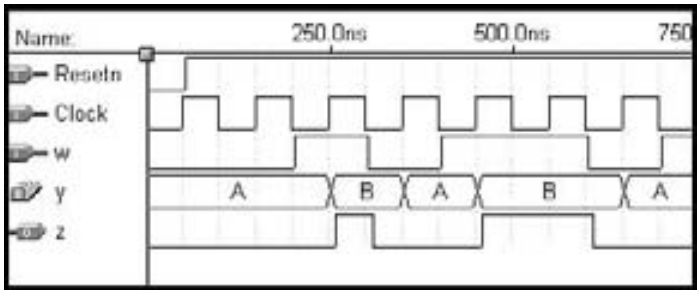


Figura 8.38 Problema potencial con entradas asíncronas a una FSM Mealy.

flanco negativo del reloj, en vez del flanco positivo cuando la FSM cambia su estado. El primer pulso en la entrada *w* dura 100 ns. Esto no debería ocasionar que la salida *z* se volviera igual a 1. Pero el circuito no se comporta así. Después que la señal *w* se vuelve 1, el primer flanco positivo del reloj hace que la FSM cambie del estado *A* al *B*. En cuanto el circuito pasa al estado *B*, la entrada *w* aún es igual a 1 durante otros 50 ns, lo que ocasiona que *z* cambie a 1. Cuando *w* regresa a 0, la señal *z* hace lo mismo. Por tanto, se genera un pulso de 50 ns erróneo en la salida *z*.

Debemos estudiar las consecuencias de este problema un poco más. Si *z* se usa para manejar otro circuito que no está controlado por el mismo reloj, entonces es probable que el pulso extraño cause grandes problemas. Pero si *z* se emplea como una entrada a un circuito (tal vez otra FSM) controlado por el mismo reloj, entonces el pulso de 50 ns será ignorado por este circuito si $z = 0$ antes del siguiente flanco positivo del reloj (contando entonces para el tiempo de preparación).

8.5 EJEMPLO DE SUMADOR SERIAL

Ahora presentaremos otro ejemplo simple que ilustra el proceso de diseño completo. En el capítulo 5 estudiamos pormenorizadamente la suma de números binarios. Explicamos varios esquemas

que pueden emplearse para sumar números de *n* bits en paralelo, que van desde sumadores con acarreo en cascada hasta sumadores con acarreo de adelanto. En estos esquemas la velocidad de la unidad sumadora es un parámetro de diseño importante. Los sumadores rápidos son más complejos y, por tanto, más costosos. Si la velocidad no es un aspecto muy signfificativo, entonces una opción económica es utilizar un *sumador serial*, en el que los bits se suman un par a la vez.

8.5.1 FSM TIPO MEALY PARA SUMADOR SERIAL

Sean $A = a_{n-1}a_{n-2}\ldots a_0$ y $B = b_{n-1}b_{n-2}\ldots b_0$ dos números sin signo que deben sumarse para producir $Sum = s_{n-1}s_{n-2}\ldots s_0$. Nuestra tarea es diseñar un circuito que realice una suma serial usando un par de bits en un ciclo de reloj. El proceso empieza con la suma de los bits a_0 y b_0. En el ciclo de reloj siguiente, los bits a_1 y b_1, incluido un posible acarreo desde la posición de bit 0, y así sucesivamente. En la figura 8.39 se muestra un diagrama de bloque de una implementación posible. Comprende tres registros de corrimiento que se usan para almacenar *A*, *B* y *Sum* conforme el cálculo avanza. Suponiendo que los registros de corrimiento de entrada tienen la capacidad de carga en paralelo, como se describe en la figura 7.19, la tarea de suma comienza cargando los valores de *A* y *B* en ellos. Entonces en cada ciclo de reloj la FSM sumadora suma un par de bits y al final del ciclo el bit de suma resultante se desplaza hacia el registro *Sum*. Utilizaremos flip-flops disparados por el flanco positivo en los que todos los cambios ocurren poco después del flanco positivo del reloj, de acuerdo con los retrasos de propagación dentro de los distintos flip-flops. En ese momento el contenido de los tres registros de corrimiento se desplaza a la derecha; el bit de suma existente en *Sum* se desplaza y se presenta el siguiente par de bits de entrada a_i y b_i a la FSM sumadora.

Ahora estamos listos para diseñar la FSM requerida. No puede ser un circuito combinacional porque deberán tomarse diferentes acciones, de acuerdo con el valor del acarreo de la posición de bit anterior. Por consiguiente se necesitan dos estados: sean *G* y *H* los estados donde los valores del acarreo de entrada son 0 y 1, respectivamente. En la figura 8.40 se presenta un diagrama de estado adecuado, definido como un modelo Mealy. El valor de salida, *s*, depende tanto del estado como del valor presente en las entradas *a* y *b*. Cada transición se etiqueta usando la notación *ab/s*, la cual indica el valor de *s* para una combinación *ab*. En el estado *G* la combinación de entrada 00 producirá $s = 0$ y la FSM permanecerá en el mismo estado. Para las combinaciones de entrada 01 y 10, la salida será $s = 1$, y la FSM permanecerá en *G*. Pero para 11,

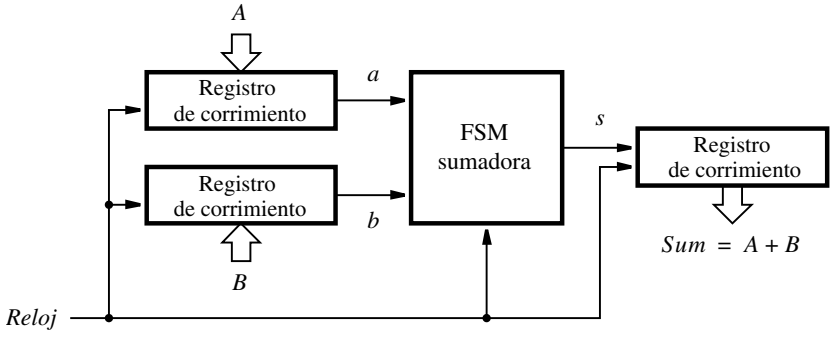
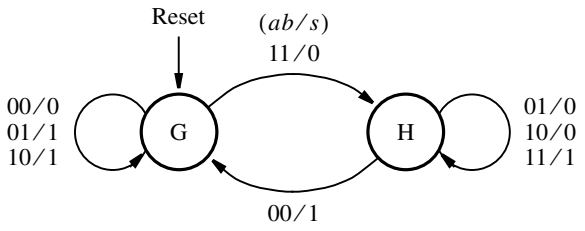


Figura 8.39 Diagrama de bloque para el sumador serial.



G: acarreo de entrada = 0
H: acarreo de entrada = 1

Figura 8.40 Diagrama de estado para la FSM sumadora serial.

se genera $s = 0$ y la máquina pasa al estado H. En éste las combinaciones 01 y 10 hacen que $s = 0$, mientras que 11 causa que $s = 1$. En los tres casos, la máquina permanece en H. Sin embargo, cuando la combinación 00 se presenta, la salida de 1 se produce y se lleva a cabo un cambio al estado G.

La tabla de estado correspondiente se presenta en la figura 8.41. Se precisa un flip-flop para representar los dos estados. La asignación de estado puede realizarse como se indicó en la figura 8.42 y conduce a las siguientes ecuaciones de estado siguiente y de salida

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

Estado presente	Estado siguiente				Salida s			
	$ab = 00$	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figura 8.41 Tabla de estado para la FSM sumadora serial.

Estado presente	Estado siguiente				Salida			
	$ab = 00$	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Figura 8.42 Tabla de asignación de estados para la figura 8.41.

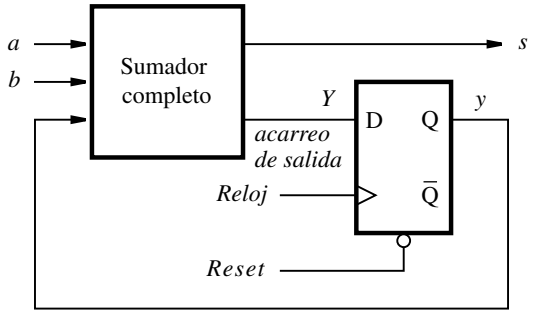


Figura 8.43 Circuito para la FSM sumadora de la figura 8.39.

Al comparar estas expresiones con las del sumador completo de la sección 5.2, es obvio que y es el acarreo de entrada, Y el de salida y s la suma del sumador completo. Por consiguiente, el cuadro de la FSM sumadora de la figura 8.39 se compone del circuito mostrado en la figura 8.43. El flip-flop puede borrarse mediante la señal *Reset* al principio de la operación.

El sumador serial es un circuito simple que puede usarse para sumar números de cualquier longitud. La estructura de la figura 8.39 está limitada en longitud por el tamaño de los registros de corrimiento.

8.5.2 FSM TIPO MOORE PARA EL SUMADOR SERIAL

En el ejemplo anterior vimos que una FSM tipo Mealy cumple bien el requisito de implementar el sumador serial. Ahora trataremos de lograr el mismo objetivo utilizando una FSM tipo Moore. Un buen punto de partida es el diagrama de estado de la figura 8.40. En una FSM tipo Moore, la salida debe depender sólo del estado de la máquina. Puesto que en los dos estados, G y H, es posible producir dos salidas según las combinaciones de las entradas a y b, una FSM tipo Moore necesitará más de dos estados. Podemos derivar un diagrama de estado adecuado dividiendo G y H en dos estados. En vez de G usaremos G_0 y G_1 para indicar que el acarreo es 0 y que la suma es ya sea 0 o 1, respectivamente. De forma similar, en vez de H emplearemos H_0 y H_1. Por tanto, la información de la figura 8.40 puede asignarse al diagrama de estado tipo Moore de la figura 8.44 de una manera sencilla.

La tabla de estado correspondiente se da en la figura 8.45 y la tabla de asignación de estados en la figura 8.46. Las expresiones de estado siguiente y de salida son

$$\begin{aligned} Y_1 &= a \oplus b \oplus y_2 \\ Y_2 &= ab + ay_2 + by_2 \\ s &= y_1 \end{aligned}$$

Las expresiones para Y_1 y Y_2 corresponden a la suma y a las expresiones de acarreo de salida en el circuito sumador completo. La FSM se implementa como se muestra en la figura 8.47. Es interesante observar que este circuito es muy parecido al de la figura 8.43. La única diferencia estriba en que en el circuito tipo Moore, la señal de salida, s, pasa por un flip-flop adicional y por tanto se retrasa un ciclo de reloj respecto al circuito secuencial. Recuérdese que

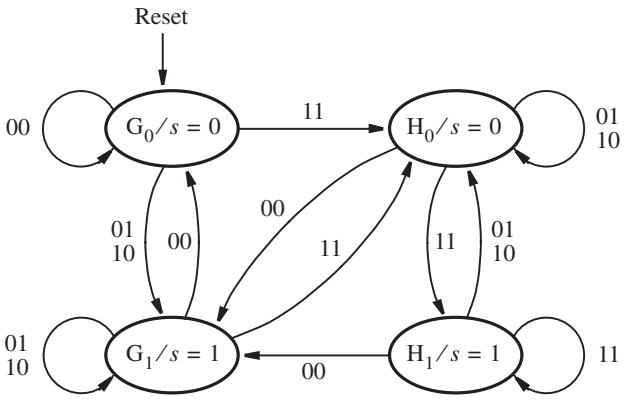


Figura 8.44 Diagrama de estado para la FSM sumadora serial tipo Moore.

Estado presente	Estado siguiente				Salida
	$ab = 00$	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figura 8.45 Tabla de estado para la FSM sumadora serial tipo Moore.

Estado presente y_2y_1	Estado siguiente				Salida
	$ab = 00$	01	10	11	s
	Y_2Y_1				
0 0	0 0	0 1	0 1	1 0	0
0 1	0 0	0 1	0 1	1 0	1
1 0	0 1	1 0	1 0	1 1	0
1 1	0 1	1 0	1 0	1 1	1

Figura 8.46 Tabla de asignación de estados para la figura 8.45.

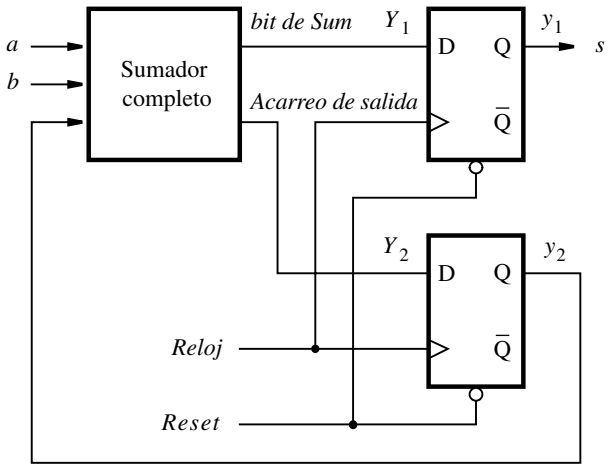


Figura 8.47 Circuito para la FSM sumadora serial tipo Moore.

observamos la misma diferencia en el ejemplo anterior, como se describió en las figuras 8.26 y 8.27.

Una diferencia sustancial entre los tipos Mealy y Moore de las FSM es que en el primero un cambio en las entradas se refleja de inmediato en las salidas, mientras que en el segundo las salidas no cambian hasta que un cambio en las entradas fuerza a la máquina a entrar en un estado nuevo, lo que ocurre un ciclo de reloj más tarde. Animamos al lector para que trace los diagramas de tiempo para los circuitos en las figuras 8.43 y 8.47, las cuales ejemplificarán posteriormente esta diferencia clave entre los dos tipos de FSM.

8.5.3 CÓDIGO DE VHDL PARA EL SUMADOR SERIAL

El sumador serial puede describirse en VHDL si se escribe código para los registros de corrimiento y la FSM sumadora. Primero diseñaremos el registro de corrimiento y luego lo utilizaremos como un subcircuito en el sumador serial.

Subcircuito del registro de corrimiento

En la figura 7.51 se proporciona el código de VHDL para un registro de corrimiento de *n* bits. En el sumador serial es beneficioso tener la capacidad de evitar que el contenido del registro de corrimiento cambie cuando ocurra un flanco activo del reloj. En la figura 8.48 aparece el código para un registro de corrimiento llamado *shiftrne*, que tiene una entrada de habilitación, *E*. Cuando $E = 1$, el registro de corrimiento se comporta del mismo modo que el de la figura 7.51. Establecer $E = 0$ impide que el contenido del registro de corrimiento cambie. La entrada *E* suele llamarse entrada *enable*. Es útil para muchos tipos de circuitos, como veremos en el capítulo 10.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

- - registro de corrimiento de izquierda a derecha con carga en paralelo y enable (habilitación)
ENTITY shiftrne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( R       : IN     STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           L, E, w : IN     STD_LOGIC ;
           Clock   : IN     STD_LOGIC ;
           Q       : BUFFER STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END shiftrne ;

ARCHITECTURE Behavior OF shiftrne IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF E = '1' THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Genbits: FOR i IN 0 TO N−2 LOOP
                    Q(i) <= Q(i+1);
                END LOOP ;
                Q(N−1) <= w ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 8.48 Código para un registro de corrimiento de izquierda a derecha con una entrada enable (habilitación).

Código completo

El código para el sumador serial se muestra en la figura 8.49. Implementa tres registros de corrimiento para las entradas *A* y *B* y la salida *Sum*. Los registros de corrimiento se cargan con datos paralelos cuando el circuito se inicializa. El diagrama de estado para la FSM sumadora se describe mediante un proceso individual, con el estilo de código de la figura 8.29. Además de los componentes del sumador serial de la figura 8.39, el código de VHDL incluye un contador descendente para determinar cuándo debe detenerse el sumador porque todos los *n* bits de la suma requerida están presentes en el registro de corrimiento de salida. Cuando el circuito se inicializa, el contador se carga con el número de bits en el sumador serial, *n*. El contador cuenta hasta 0 y luego se detiene e inhabilita los cambios posteriores en el registro de corrimiento de salida.

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;

3  ENTITY serial IS
4      GENERIC ( length : INTEGER := 8 ) ;
5      PORT ( Clock  : IN         STD_LOGIC ;
6             Reset  : IN         STD_LOGIC ;
7             A, B   : IN         STD_LOGIC_VECTOR(length−1 DOWNTO 0) ;
8             Sum    : BUFFER     STD_LOGIC_VECTOR(length−1 DOWNTO 0) );
9  END serial ;

10 ARCHITECTURE Behavior OF serial IS
11     COMPONENT shiftrne
12         GENERIC ( N : INTEGER := 4 ) ;
13         PORT( R        : IN         STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
14               L, E, w  : IN         STD_LOGIC ;
15               Clock    : IN         STD_LOGIC ;
16               Q        : BUFFER     STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
17     END COMPONENT ;

18     SIGNAL QA, QB, Null_in : STD_LOGIC_VECTOR(length−1 DOWNTO 0) ;
19     SIGNAL s, Low, High, Run : STD_LOGIC ;
20     SIGNAL Count : INTEGER RANGE 0 TO length ;
21     TYPE State_type IS (G, H) ;
22     SIGNAL y : State_type ;

       …continúa en el inciso b
```

Figura 8.49 Código de VHDL para el sumador serial (inciso *a*).

Las líneas de código de la figura 8.49 están numeradas a la izquierda para facilitar las referencias. El parámetro GENERIC *length* establece el número de bits en el sumador serial. Como el valor de *length* es igual a 8, el código representa un sumador serial para números de ocho bits. Al cambiar el valor de *length*, el mismo código sirve para sintetizar un circuito de sumador serial para cualquier número de bits.

Las líneas 18 a 22 definen varias señales utilizadas en el código. Las señales Q*A* y Q*B* corresponden a las salidas en paralelo de los registros de corrimiento con entradas *A* y *B* de la figura 8.39. La señal llamada *s* representa la salida de la FSM sumadora. Las otras señales se describirán junto con las líneas de código cuando se usen.

En la figura 8.39 los registros de corrimiento para las entradas *A* y *B* no emplean una entrada serial ni una entrada enable. Sin embargo, el componente *shiftrne*, el cual se utiliza para los tres registros de corrimiento, incluye estos puertos y por tanto las señales deben conectarse a ellos. La entrada enable para los dos registros de corrimiento puede conectarse al valor lógico 1. El valor desplazado en la entrada serial no importa, así que puede conectarse tanto a 1 como a 0. En

```
23 BEGIN
24      Low <= '0' ; High <= '1' ;
25      ShiftA: shiftrne GENERIC MAP (N => length)
26           PORT MAP ( A, Reset, High, Low, Clock, QA ) ;
27      ShiftB: shiftrne GENERIC MAP (N => length)
28           PORT MAP ( B, Reset, High, Low, Clock, QB ) ;
29      AdderFSM: PROCESS ( Reset, Clock )
30      BEGIN
31           IF Reset = '1' THEN
32                y <= G ;
33           ELSIF Clock'EVENT AND Clock = '1' THEN
34                CASE y IS
35                     WHEN G =>
36                          IF QA(0) = '1' AND QB(0) = '1' THEN y <= H ;
37                          ELSE y <= G ;
38                          END IF ;
39                     WHEN H =>
40                          IF QA(0) = '0' AND QB(0) = '0' THEN y <= G ;
41                          ELSE y <= H ;
42                          END IF ;
43                END CASE ;
44           END IF ;
45      END PROCESS AdderFSM ;

46      WITH y SELECT
47           s <=  QA(0) XOR QB(0) WHEN G,
48                  NOT ( QA(0) XOR QB(0) ) WHEN H ;
49      Null_in <= (OTHERS => '0') ;
50      ShiftSum: shiftrne GENERIC MAP ( N => length )
51                          PORT MAP ( Null_in, Reset, Run, s, Clock, Sum ) ;
52      Stop: PROCESS
53      BEGIN
54           WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
55           IF Reset = '1' THEN
56                Count <= length ;
57           ELSIF Run = '1' THEN
58                Count <= Count −1 ;
59           END IF ;
60      END PROCESS ;
61      Run <= '0' WHEN Count = 0 ELSE '1' ; - - detiene el contador y ShiftSum
62 END Behavior ;
```

Figura 8.49 Código de VHDL para el sumador serial (inciso *b*).

las líneas 26 y 28, la entrada enable está conectada a la señal llamada *High*, que se establece en 1, y las entradas seriales se vinculan a la señal *Low*, que es 0. Estas señales son necesarias porque la sintaxis de VHDL no permite que las constantes 0 o 1 se conecten a los puertos de un componente. El parámetro *n* para cada registro de corrimiento se establece en *length* usando GENERIC MAP. Si GENERIC MAP no se proporcionara, entonces se utilizaría el valor predeterminado de $N = 4$ dado en el código de la figura 8.48. Los registros de corrimiento se cargan en paralelo por medio de la señal *Reset*. Hemos elegido usar una señal reset activa en nivel alto para el circuito.

La FSM sumadora se especifica en las líneas 29 a 45, donde se describen las transiciones de estado de la figura 8.41. En las líneas 46 a 48 se define la salida, *s*, de la FSM sumadora. Esta instrucción es resultado de observar en la figura 8.41 que cuando la FSM se halla en el estado *G*, la suma es $s = a \oplus b$, y cuando está en el estado *H*, la suma es $s = \overline{a \oplus b}$.

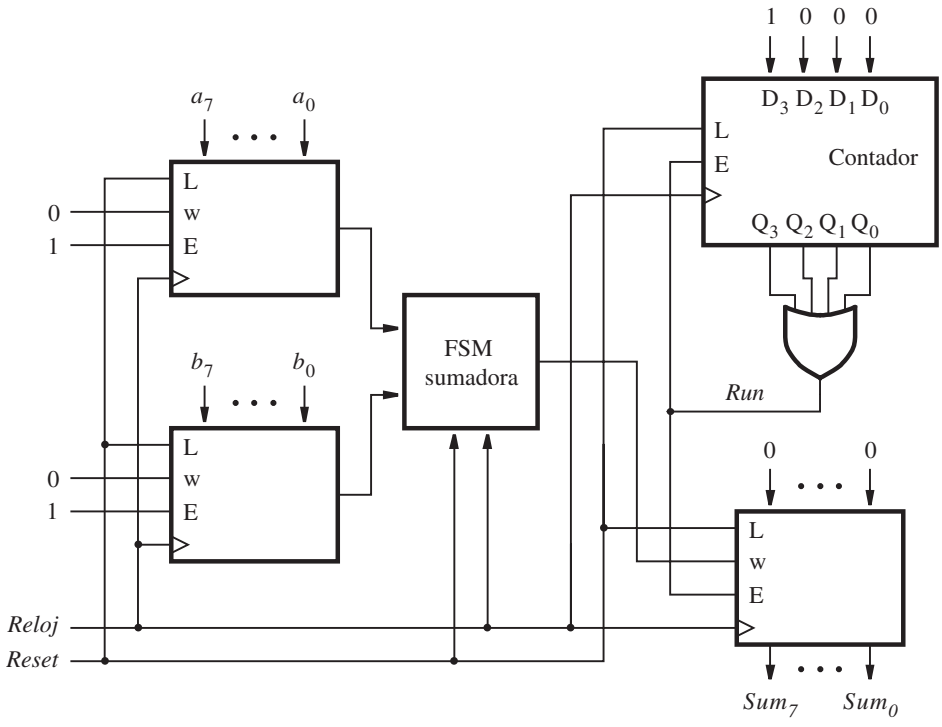
El registro de corrimiento de salida no necesita una entrada de datos en paralelo. Sin embargo, como el componente *shiftrne* tiene este puerto una señal debe conectarse a él. La señal llamada *Null_in* se usa para tal fin. En la línea 49 se establece *Null_in*, que es una señal STD_LOGIC_VECTOR, en 0. El número de bits de esta señal se define mediante la constante *length*. Por consiguiente, no podemos usar la sintaxis normal de VHDL, una cadena de ceros (0) encerrada con comillas, para establecer todos los bits en 0. Una solución a este problema consiste en emplear la sintaxis (OTHERS => '0'), la cual explicamos en la exposición referente a la figura 7.46. La entrada enable para el registro de corrimiento se llama *Run*. Se deriva de las salidas del contador descendente descrito en las líneas 52 a 60. Cuando *Reset* = 1, *Count* se inicializa en el valor de *length*. Por tanto, siempre que *Run* = 1, *Count* disminuye en cada ciclo de reloj. En la línea 61 *Run* se establece en 0 cuando *Count* es igual a 0. Nótese que no se utilizan comillas en la condición *Count* = 0 porque el 0 sin comillas tiene el tipo entero.

Síntesis y simulación del código de VHDL

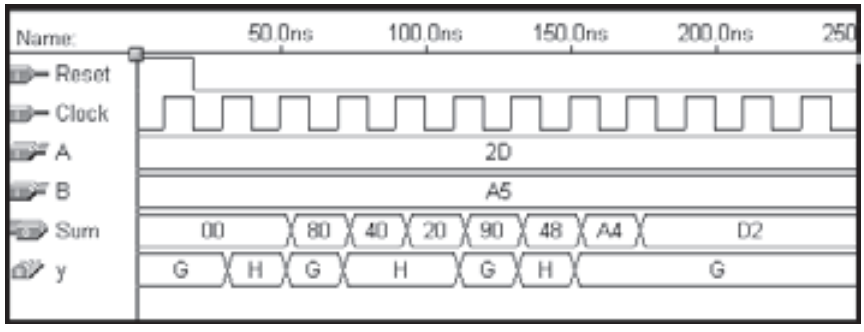
Los resultados de sintetizar un circuito a partir del código de la figura 8.49 se ilustran en la figura 8.50*a*. A las salidas del contador se les aplica una compuerta OR para proporcionar la señal *Run*, la cual habilita la sincronización tanto del registro de corrimiento de salida como del contador. Un ejemplo de una simulación de sincronización para el circuito se muestra en la figura 8.50*b*. Primero se inicializa el circuito, lo que da como resultado los valores de *A* y *B* que se cargan en los registros de corrimiento, y el valor de *length* (8) se carga en el contador descendente. Después de cada ciclo de reloj, la FSM sumadora suma un par de bits de los números de entrada, y el bit de suma se desplaza hacia el registro de corrimiento de salida. Después de ocho ciclos de reloj el registro de corrimiento de salida contiene la suma correcta y el corrimiento se detiene haciendo que la señal *Run* se vuelva 0.

8.6 MINIMIZACIÓN DE ESTADOS

Nuestros ejemplos introductorios de máquinas de estado finito son tan simples que es fácil ver que el número de estados que usamos fue el mínimo posible para realizar la función requerida. Cuando un diseñador ha de diseñar una FSM más compleja, es probable que el intento inicial resulte en una máquina con más estados de los que en realidad se necesitan. Reducir el número de estados al mínimo nos interesa porque tal vez se precisen menos flips-flops para representar los estados y quizá se reduzca la complejidad del circuito combinacional que se necesita en la FSM.



a) Circuito



b) Resultados de la simulación

Figura 8.50 Sumador serial sintetizado.

Si el número de estados en una FSM puede reducirse, entonces algunos estados en el diseño original deben equivaler a otros estados en su contribución al comportamiento general de la FSM. Podemos expresar esta idea de una manera más formal con la definición siguiente.

Definición 8.1 *Se dice que dos estados S_i y S_j son equivalentes si y sólo si para cada secuencia de entrada posible, se produce la misma secuencia de salida independientemente que S_i o S_j sea el valor inicial.*

Es posible definir un procedimiento de minimización que busque cualesquiera estados equivalentes. Resulta muy tedioso realizar manualmente un procedimiento como éste, pero puede automatizarse para usarlo en las herramientas CAD. No lo estudiaremos aquí porque es aburrido. Sin embargo, para dar una idea del efecto de la minimización del estado, presentaremos otro enfoque, que resulta mucho más eficiente pero no tan amplio en posibilidades.

En vez de tratar de mostrar que algunos estados en una FSM son equivalentes, a menudo es más fácil mostrar que algunos estados definitivamente **no** lo son. Esta idea puede aprovecharse para definir un procedimiento de minimización simple.

8.6.1 PROCEDIMIENTO DE MINIMIZACIÓN POR PARTICIÓN

Supóngase que una máquina de estado tiene una sola entrada w. Entonces, si la señal de entrada $w = 0$ se aplica a esta máquina en el estado S_i y el resultado es que la máquina cambia al estado S_u, diremos que S_u es un *sucesor* 0 de S_i. De modo similar, si $w = 1$ se aplica en el estado S_i y esto ocasiona que la máquina cambie al estado S_v, diremos que S_v es un *sucesor* 1 de S_i. En general, nos referiremos a los sucesores de S_i como sus sucesores k. Cuando la FSM tiene sólo una entrada, k puede ser 0 o 1. Pero si hay varias entradas a la FSM, entonces k representa el conjunto de todas las combinaciones posibles de las entradas.

Con base en la definición 8.1 se deduce que si los estados S_i y S_j son equivalentes, entonces sus sucesores k correspondientes (para todas las k) también son equivalentes. A partir de este hecho podemos plantear un procedimiento de minimización que suponga considerar los estados de la máquina como un conjunto que luego se divide en *particiones* que comprendan los subconjuntos que definitivamente no son equivalentes.

Definición 8.2 *Una partición consta de uno o más bloques, donde cada bloque comprende un subconjunto de estados que pueden ser equivalentes, pero los estados de un bloque definitivamente no son equivalentes a los de otros.*

Supongamos de inicio que todos los estados son equivalentes; esto forma la partición inicial, P_1, en la que todos los estados están en el mismo bloque. Como paso siguiente, formaremos la partición, P_2, en la cual el conjunto de estados se parte de tal manera que los estados de cada bloque generen los mismos valores de salida. Obviamente, no es posible que los estados que generan diferentes salidas sean equivalentes. Por tanto seguiremos formulando particiones nuevas probando si los sucesores k de los estados de cada bloque están contenidos en un bloque. Dichos estados cuyos sucesores k están en diferentes bloques no pueden estar en un bloque. De esta forma, en cada nueva partición se forman bloques nuevos. El proceso termina cuando una nueva partición es igual a la anterior. En consecuencia, todos los estados en un bloque cualquiera son equivalentes. Para ilustrar el procedimiento considérese el ejemplo 8.5.

Ejemplo 8.5

En la figura 8.51 se muestra una tabla de estado para una FSM en particular. En un intento por minimizar el número de estados, apliquemos el procedimiento de partición. La partición inicial contiene todos los estados de un solo bloque

$$P_1 = (ABCDEFG)$$

La partición siguiente separa los estados que tienen diferentes salidas (observe que esta FSM es del tipo Moore), lo que significa que los estados A, B y D deben ser diferentes de los estados C, E, F y G. Por ende, la nueva partición tiene dos bloques

$$P_2 = (ABD)(CEFG)$$

Ahora debemos considerar todos los sucesores 0 y 1 de los estados de cada bloque. Para el bloque (ABD), los sucesores 0 son (BDB), respectivamente. Como todos estos estados sucesores están en el mismo bloque en P_2, aún debemos suponer que los estados A, B y D pueden ser equivalentes. Los sucesores 1 para estos estados son (CFG). Puesto que estos sucesores se hallan también en el mismo bloque en P_2, concluimos que (ABD) debe permanecer en un bloque de P_3. Enseguida considérese el bloque ($CEFG$). Sus sucesores 0 son ($FFEF$), respectivamente. Están en el mismo bloque en P_2. Los sucesores 1 son ($ECDG$). Como estos estados no se hallan en el mismo bloque en P_2, cuando menos uno de los estados del bloque ($CEFG$) no es equivalente a los otros. En particular, el estado F debe ser diferente de los estados C, E y G, ya que su sucesor 1 es D, el cual está en un bloque diferente de C, E y G. Por consiguiente

$$P_3 = (ABD)(CEG)(F)$$

Al repetir el proceso se produce lo siguiente. Los sucesores 0 de (ABD) son (BDB), los cuales se hallan en el mismo bloque de P_3. Los sucesores 1 son (CFG), que no están en el mismo bloque. Puesto que F se encuentra en un bloque distinto que C y G, se deduce que el estado B no puede ser equivalente a los estados A y D. Los sucesores 0 y 1 de (CEG) son (FFF) y (ECG), respectivamente. Los dos subconjuntos se acomodan en los bloques de P_3. Por ende,

$$P_4 = (AD)(B)(CEG)(F)$$

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Figura 8.51 Tabla de estado para el ejemplo 8.5.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

Figura 8.52 Tabla de estados mínimos para el ejemplo 8.5.

Si seguimos el mismo enfoque para verificar los sucesores 0 y 1 de los bloques (*AD*) y (*CEG*) encontramos que

$$P_5 = (AD)(B)(CEG)(F)$$

Como $P_5 = P_4$ y no se genera ningún bloque nuevo, se deduce que los estados de cada bloque son equivalentes. Si los estados en algunos bloques no lo son, entonces sus sucesores *k* tendrían que estar en bloques distintos. Por tanto, los estados *A* y *D* son equivalentes, y *C*, *E* y *G* son equivalentes. Como cada bloque puede representarse por medio de un solo estado, sólo se precisan cuatro estados para implementar la FSM definida por la tabla de estados de la figura 8.51. Si el símbolo *A* representa los dos estados *A* y *D* de la figura y el símbolo *C* denota los estados *C*, *E* y *G*, entonces la tabla de estados se reduce a la de la figura 8.52.

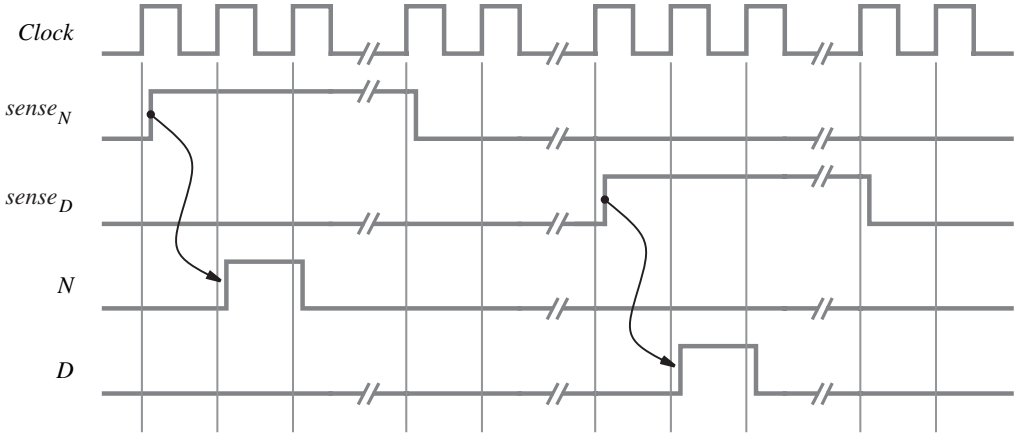
El efecto de la minimización es que hemos encontrado una solución que requiere sólo dos flip-flops para realizar los cuatro estados de la tabla de estados minimizada, en vez de necesitar tres flip-flops para el diseño original. Aunque la expectativa es que será más sencillo implementar la FSM con menos estados, no siempre es así.

El concepto de minimización de estados se basa en el hecho de que dos FSM diferentes pueden presentar un comportamiento idéntico en términos de las salidas producidas en respuesta a todas las entradas posibles. Estas máquinas son funcionalmente equivalentes, aun cuando se implementan con circuitos que pueden ser muy distintos. En general, no es fácil determinar si dos FSM cualesquiera son equivalentes o no. Nuestro procedimiento de minimización asegura que una FSM simplificada es funcionalmente equivalente a la original. Invitamos al lector a reflexionar respecto a que las FSM de las figuras 8.51 y 8.52 son funcionalmente equivalentes al implementar ambas máquinas y simular su comportamiento utilizando las herramientas CAD.

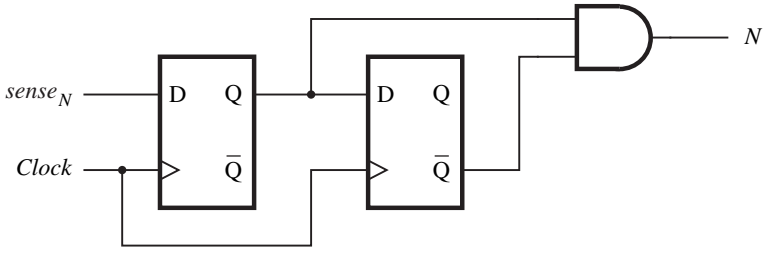
Ejemplo 8.6 Como otro ejemplo de minimización consideremos el diseño de un circuito secuencial práctico que podría utilizarse en una máquina expendedora. Supóngase que una máquina expendedora operada por monedas despacha caramelos de acuerdo con ciertas condiciones:

- La máquina acepta monedas de 5 y 10 centavos.
- Se requieren 15 centavos para que un caramelo sea expulsado de la máquina.
- Si se depositan 20 centavos, la máquina no devolverá cambio, pero otorgará un crédito de 5 centavos al comprador y esperará que éste haga una segunda compra.

Todas las señales electrónicas de la máquina expendedora están sincronizadas con un flanco positivo de una señal de reloj, llamada *Clock*. La frecuencia exacta de la señal de reloj no es importante para nuestro ejemplo, pero supondremos un periodo de reloj de 100 ns. El mecanismo receptor de monedas de la máquina expendedora genera dos señales, $sense_D$ y $sense_N$, las cuales se validan cuando se detecta una moneda de 10 o de 5 centavos. El receptor de monedas es un dispositivo mecánico y por tanto es muy lento en comparación con un circuito electrónico, así que insertar una moneda hace que $sense_D$ o $sense_N$ se establezcan en 1 durante un gran número de ciclos de reloj. Supondremos que el receptor de monedas también genera otras dos señales: *D* y *N*. La señal *D* se establece en 1 para un ciclo de reloj después que $sense_D$ se vuelve 1 y *N* se establece en 1 para un ciclo de reloj después que $sense_N$ se vuelve 1. Las relaciones de tiempo entre *Clock*, $sense_D$, $sense_N$, *D* y *N* se ilustran en la figura 8.53*a*. Las líneas quebradas en las formas de onda indican que $sense_D$ o $sense_N$ pueden ser 1 durante muchos ciclos de reloj. Además, puede haber un periodo arbitrariamente largo entre la inserción de dos monedas consecutivas. Observe que como el receptor de monedas puede aceptar sólo una a la vez, no es posible tener ambas, *D* y *N*, en 1 a la vez. En la figura 8.53*b* se ilustra cómo puede generarse la señal *N* a partir de la señal $sense_N$.



a) Diagrama de tiempo



b) Circuito que genera *N*

Figura 8.53 Señales para la máquina expendedora.

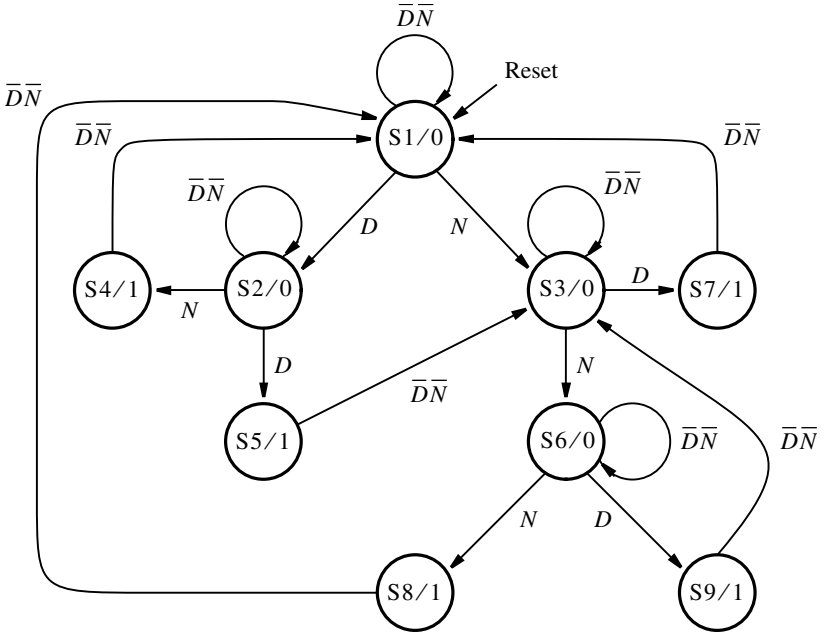


Figura 8.54 Diagrama de estado para el ejemplo 8.6.

Con base en estas suposiciones, podemos elaborar un diagrama de estado inicial de una manera muy sencilla, como se indica en la figura 8.54. Las entradas a la FSM son D y N, y el estado inicial es $S1$. Mientras $D = N = 0$, la máquina permanece en el estado $S1$, lo cual se indica por medio del arco etiquetado $\overline{D} \cdot \overline{N} = 1$. La inserción de una moneda de 10 centavos conduce al estado $S2$, mientras que insertar una de 5 centavos lleva al estado $S3$. En ambos casos el monto depositado es menor que 15 centavos, lo que no es suficiente para soltar el caramelo. Esto se indica por la salida z, igual a 0, como en $S2/0$ y $S3/0$. La máquina permanecerá en el estado $S2$ o $S3$ hasta que otra moneda se deposite, ya que $D = N = 0$. En el estado $S2$ una moneda de cinco centavos ocasionará una transición a $S4$ y una moneda de 10 centavos, a $S5$. En estos dos estados, se deposita suficiente dinero para activar el mecanismo de salida que suelta el caramelo; por consiguiente, los nodos de estado tienen las etiquetas $S4/1$ y $S5/1$. En $S4$ la cantidad depositada es 15 centavos, lo que significa que en el siguiente flanco activo del reloj la máquina debe volver al estado inicial $S1$. La condición $\overline{D} \cdot \overline{N}$ en el arco que sale de $S4$ está garantizada para ser verdadera porque la máquina permanece en el estado $S4$ sólo durante 100 ns, que es muy poco tiempo para que una nueva moneda se haya depositado.

El estado $S5$ denota que se depositó una cantidad de 20 centavos. El caramelo se suelta y en el siguiente flanco del reloj la FSM hace una transición al estado $S3$, el cual representa un crédito de 5 centavos. Un razonamiento similar cuando la máquina se halla en el estado $S3$ lleva a los estados $S6$ a $S9$. Esto completa el diagrama de estado para la FSM deseada. Una versión de la tabla de estado de la misma información se presenta en la figura 8.55.

Observe que la condición $D = N = 1$ se indica como condición no-importa en la tabla. Note también otras condiciones no-importa en los estados $S4$, $S5$, $S7$, $S8$ y $S9$. Éstas corresponden a

Estado presente	Estado siguiente				Salida
	DN = 00	01	10	11	z
S1	S1	S3	S2	—	0
S2	S2	S4	S5	—	0
S3	S3	S6	S7	—	0
S4	S1	—	—	—	1
S5	S3	—	—	—	1
S6	S6	S8	S9	—	0
S7	S1	—	—	—	1
S8	S1	—	—	—	1
S9	S3	—	—	—	1

Figura 8.55 Tabla de estado para el ejemplo 8.6.

los casos donde no hay necesidad de revisar las señales D y N porque la máquina cambia a otro estado en un periodo muy breve para que una nueva moneda se haya insertado.

Por medio del procedimiento de minimización obtenemos las particiones siguientes

$$P_1 = (S1, S2, S3, S4, S5, S6, S7, S8, S9)$$
$$P_2 = (S1, S2, S3, S6)(S4, S5, S7, S8, S9)$$
$$P_3 = (S1)(S3)(S2, S6)(S4, S5, S7, S8, S9)$$
$$P_4 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$
$$P_5 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$

La partición final tiene cinco bloques. Sea $S2$ la equivalencia a $S6$, sea $S4$ lo mismo respecto a $S7$ y $S8$, y sea $S5$ equivalente a $S9$. Esto conduce a la tabla de estado minimizada de la figura 8.56. El circuito que implementa esta tabla puede estar diseñado como se explicó en las secciones anteriores.

Estado presente	Estado siguiente				Salida
	DN = 00	01	10	11	z
S1	S1	S3	S2	—	0
S2	S2	S4	S5	—	0
S3	S3	S2	S4	—	0
S4	S1	—	—	—	1
S5	S3	—	—	—	1

Figura 8.56 Tabla de estado minimizada para el ejemplo 8.6.

En este ejemplo utilizamos un método directo para derivar el diagrama de estado original, que luego minimizamos utilizando el procedimiento de particionamiento. En la figura 8.57 se presenta la información de la tabla de estado de la figura 8.56 en forma de un diagrama de estado. Al observar este diagrama, es probable que el lector vea que pudo haber sido más fácil derivar directamente el diagrama mejorado usando el siguiente razonamiento. Suponga que los estados corresponden a las distintas cantidades de dinero depositado. En particular, los estados *S*1, *S*3, *S*2, *S*4 y *S*5 corresponden a las cantidades 0, 5, 10, 15 y 20 centavos, respectivamente. Con esta interpretación de los estados no es difícil derivar los arcos de transición que definen la FSM. En la práctica, el diseñador a menudo puede producir los diseños iniciales que no tienen una gran cantidad de estados superfluos.

Hemos encontrado una solución que requiere cinco estados, que es el número mínimo de estados para una FSM tipo Moore que realiza la tarea buscada para controlar la venta. En la sección 8.3 aprendimos que las FSM tipo Mealy pueden necesitar menos estados que las máquinas tipo Moore, aun cuando no necesariamente conducen a implementaciones generales más simples. Si usamos el modelo Mealy podemos eliminar los estados *S*4 y *S*5 de la figura 8.57. El resultado se muestra en la figura 8.58. Esta versión requiere sólo tres estados, pero las funciones de salida se tornan más complicadas. Se invita al lector a comparar la complejidad de las implementaciones completando los pasos de diseño para las FSM de las figuras 8.57 y 8.58.

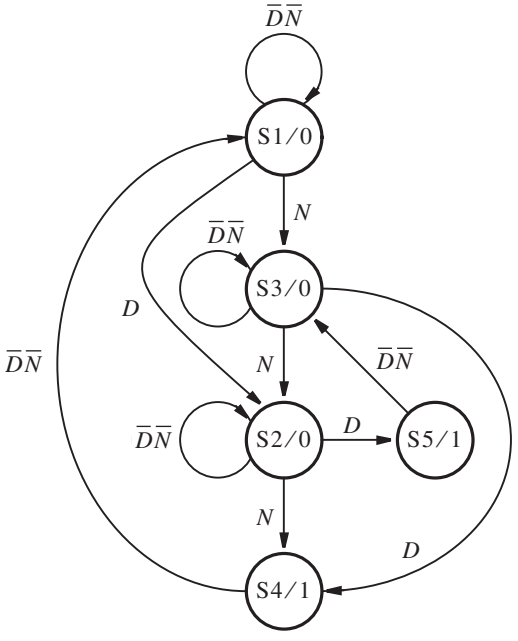


Figura 8.57 Diagrama de estado minimizado para el ejemplo 8.6.

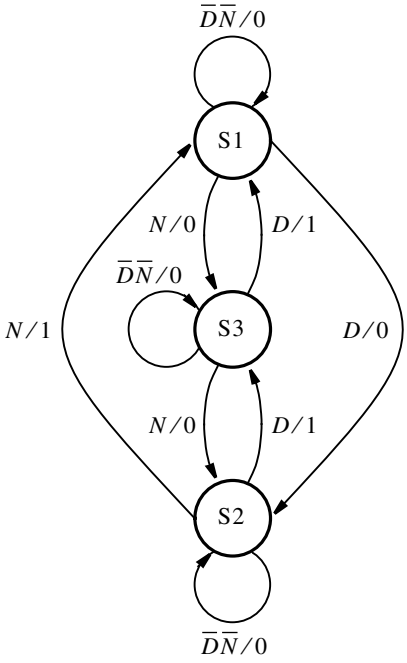


Figura 8.58 FSM tipo Mealy para el ejemplo 8.6.

8.6.2 FSM ESPECIFICADAS DE MANERA INCOMPLETA

El esquema de particionamiento para la minimización de estados funciona bien cuando se especifican todas las entradas de la tabla de estado. Tal es el caso de la FSM definida en la figura 8.51. Se dice que las FSM de este tipo están *especificadas por completo*. Si una o más entradas en la tabla de estado no están especificadas, y corresponden a condiciones no-importa, entonces se dice que la FSM está *especificada de manera incompleta*. Un ejemplo de una FSM como ésta se presenta en la figura 8.55. Como vimos en el ejemplo 8.6, el esquema de particionamiento también funciona bien para esta FSM. Pero en general, ese esquema es menos útil cuando se involucran FSM especificadas de manera incompleta, como se ilustra en el ejemplo 8.7.

Ejemplo 8.7

Considere la FSM de la figura 8.59, la cual tiene cuatro entradas sin especificar, ya que hemos supuesto que la entrada $w = 1$ no ocurrirá cuando la máquina se halle en los estados B o G. En consecuencia, ni una transición de estado ni un valor de salida se especifica para estos dos casos. Una diferencia importante entre esta FSM y la de la figura 8.55 es que algunas salidas en esta FSM no están especificadas, mientras que en la otra FSM todas las salidas lo están.

El procedimiento de minimización por particionamiento puede aplicarse a las FSM tipo Mealy de la misma forma que para las FSM tipo Moore de los ejemplos 8.5 y 8.6. Dos estados

Estado presente	Estado siguiente		Salida z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	—	0	—
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	—	0	—

Figura 8.59 Tabla de estado especificada de manera incompleta para el ejemplo 8.7.

se consideran equivalentes y, por tanto, se colocan en el mismo bloque de una partición, si sus salidas son iguales para todas las combinaciones de entrada correspondientes. Para realizar el proceso de particionamiento podemos suponer que las salidas sin especificar tienen un valor específico. Sin saber si estos valores deben ser 0 o 1, supongamos primero que las dos salidas sin especificar tienen un valor de 0. Entonces las primeras dos particiones son

$$P_1 = (ABCDEFG)$$
$$P_2 = (ABDG)(CEF)$$

Observe que los estados A, B, D y G están en el mismo bloque porque sus salidas son iguales a 0 para $w = 0$ y $w = 1$. Además, los estados C, E y F se encuentran en un bloque pues presentan el mismo comportamiento de salida; todos generan $z = 0$ si $w = 0$, y $z = 1$ si $w = 1$. Si se continúa con el procedimiento de partición se obtienen las particiones restantes

$$P_3 = (AB)(D)(G)(CE)(F)$$
$$P_4 = (A)(B)(D)(G)(CE)(F)$$
$$P_5 = P_4$$

El resultado es una FSM especificada por seis estados.

A continuación considere la alternativa de suponer que las dos salidas sin especificar de la figura 8.59 tienen un valor de 1. Esto nos llevaría a las particiones

$$P_1 = (ABCDEFG)$$
$$P_2 = (AD)(BCEFG)$$
$$P_3 = (AD)(B)(CEFG)$$
$$P_4 = (AD)(B)(CEG)(F)$$
$$P_5 = P_4$$

Esta solución supone cuatro estados. Obviamente, la elección de los valores asignados a las salidas sin especificar tiene gran importancia.

No estudiaremos más el tema de la minimización de estados de las FSM especificadas de manera incompleta. Como acabamos de mencionar, es posible desarrollar una técnica de minimización que busque estados equivalentes con base directa en la definición 8.1. Este enfoque se describe en muchos libros de diseño lógico [2, 5-8, 12-14].

Por último, es importante aclarar que reducir el número de estados en una FSM no necesariamente lleva a una implementación más simple. Resulta de interés que el efecto de la asignción de estados, analizada en la sección 8.2, puede tener una mayor influencia en la simplicidad de la implementación que la influencia que tiene la minimización de estados. En un entorno de diseño moderno, el diseñador se basa en las herramientas CAD para implementar máquinas de estado de una manera eficiente.

8.7 DISEÑO DE UN CONTADOR UTILIZANDO EL ENFOQUE DEL CIRCUITO SECUENCIAL

En esta sección estudiaremos el diseño de un circuito contador empleando el enfoque general para diseñar circuitos secuenciales. Del capítulo 7 sabemos que los contadores pueden realizarse como etapas en cascada de flip-flops y algunas compuertas lógicas, donde cada etapa divide entre dos el número de pulsos que llegan. Para mantener nuestro ejemplo simple, elegimos un contador de tamaño pequeño pero también mostramos cómo el diseño puede ampliarse a tamaños más grandes. La especificación para el contador es

- La secuencia de conteo es 0, 1, 2,..., 6, 7, 0, 1,...
- Existe una señal de entrada w, cuyo valor se considera durante cada ciclo de reloj. Si $w = 0$, el conteo presente aún es el mismo; si $w = 1$, el conteo se incrementa.

El contador puede designarse como un circuito secuencial síncrono mediante las técnicas presentadas en las secciones anteriores. Mostramos primero el enfoque manual clásico para diseñar el contador, el cual ilustra los conceptos básicos inherentes al proceso de diseño. Después, se indica cómo se logra la tarea de diseño con las herramientas CAD, lo que es mucho más fácil de hacer y señala cómo se haría lo mismo en la práctica.

8.7.1 DIAGRAMA DE ESTADO Y TABLA DE ESTADO PARA UN CONTADOR MÓDULO 8

En la figura 8.60 aparece un diagrama de estado para el contador buscado. Hay un estado asociado con cada conteo. En el diagrama el estado A corresponde al conteo 0, el estado B al conteo 1 y así sucesivamente. Mostramos las transiciones entre los estados necesarias para implementar la secuencia de conteo. Obsérvese que las señales de salida se especifican como dependientes sólo del estado del contador en un momento dado, el cual es el modelo Moore de los circuitos secuenciales.

El diagrama de estado puede representarse en forma de tabla de estado, como se muestra en la figura 8.61.

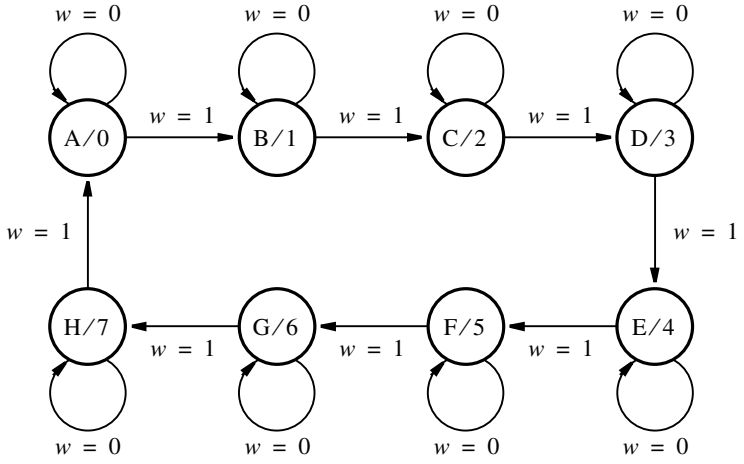


Figura 8.60 Diagrama de estado para el contador.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figura 8.61 Tabla de estado para el contador.

8.7.2 ASIGNACIÓN DE ESTADOS

Se necesitan tres variables de estado para representar los ocho estados. Sean estas variables y_2, y_1 y y_0, las cuales denotan el estado presente. Sean Y_2, Y_1 y Y_0 las funciones correspondientes al estado siguiente. La asignación de estado más práctica (y simple) consiste en codificar cada estado con el número binario que el contador debe dar como salida en ese estado. Las señales de salida requeridas serán iguales a las señales que representan las variables de estado. Esto conduce a la tabla de asignación de estados de la figura 8.62.

El paso final del diseño consiste en elegir el tipo de flip-flops y derivar la expresión que controla sus entradas. La opción más sencilla es utilizar flip-flops tipo D. Primero seguimos este método.

	Estado presente $y_2y_1y_0$	Estado siguiente $w = 0$ $Y_2Y_1Y_0$	Estado siguiente $w = 1$ $Y_2Y_1Y_0$	Conteo $z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figura 8.62 Tabla de asignación de estados para el contador.

Luego mostramos la alternativa de utilizar flip-flops JK. En cualquier caso, los flip-flops deben dispararse por flanco para asegurar que sólo ocurre una transición durante un ciclo de reloj.

8.7.3 IMPLEMENTACIÓN UTILIZANDO FLIP-FLOPS D

Cuando se usan flip-flops D para realizar la máquina de estado finito, cada función del estado siguiente, Y_i, está conectada a la entrada D del flip-flop que implementa la variable de estado y_i. Las funciones del estado siguiente se derivan de la información de la figura 8.62. Con los mapas de Karnaugh de la figura 8.63 obtenemos la implementación siguiente

$$D_0 = Y_0 = \overline{w}y_0 + w\overline{y}_0$$
$$D_1 = Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$$
$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$$

El circuito resultante se muestra en la figura 8.64. No resulta aparente cómo ampliar este circuito para implementar un contador más grande porque no hay un patrón de borrado claro en las expresiones para D_0, D_1 y D_2. Sin embargo, podemos volver a escribir estas expresiones como sigue

$$\begin{aligned} D_0 &= \overline{w}y_0 + w\overline{y}_0 \\ &= w \oplus y_0 \end{aligned}$$

$$\begin{aligned} D_1 &= \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1 \\ &= (\overline{w} + \overline{y}_0)y_1 + wy_0\overline{y}_1 \\ &= \overline{w\overline{y}_0}y_1 + wy_0\overline{y}_1 \\ &= wy_0 \oplus y_1 \end{aligned}$$

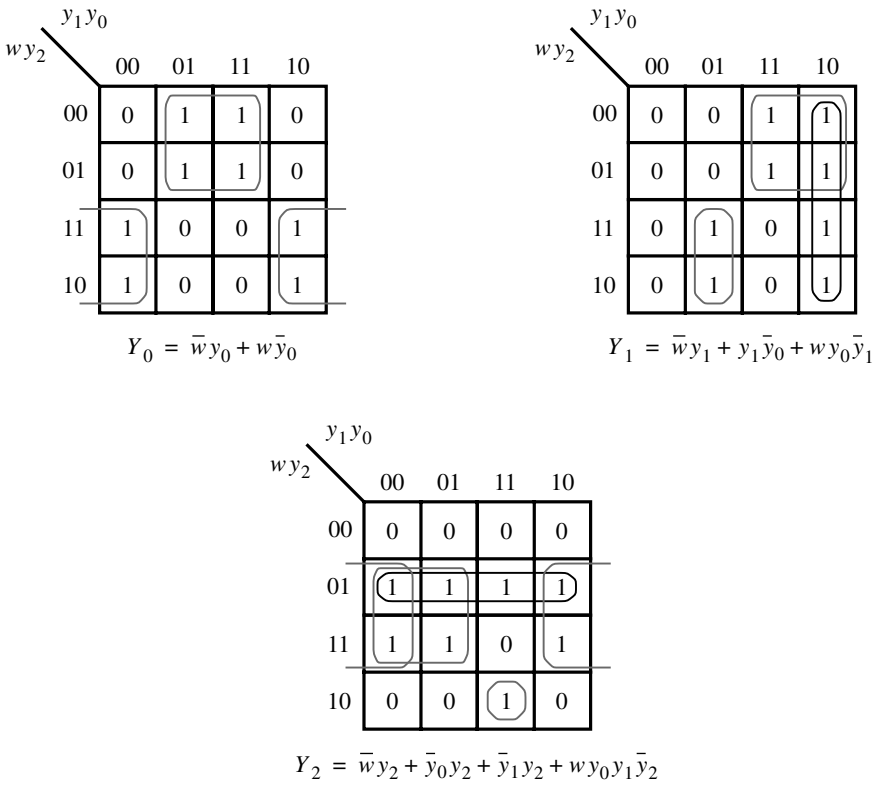


Figura 8.63 Mapas de Karnaugh para los flip-flops D para el contador.

$$\begin{aligned} D_2 &= \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2 \\ &= (\overline{w} + \overline{y}_0 + \overline{y}_1)y_2 + wy_0y_1\overline{y}_2 \\ &= \overline{wy_0y_1}y_2 + wy_0y_1\overline{y}_2 \\ &= wy_0y_1 \oplus y_2 \end{aligned}$$

Entonces ya aparece un patrón obvio, que nos lleva al circuito de la figura 7.24.

8.7.4 IMPLEMENTACIÓN UTILIZANDO FLIP-FLOPS JK

Los flip-flops JK brindan una posibilidad atractiva. Si se emplean para implementar el circuito secuencial especificado en la figura 8.62 se requiere la derivación de las entradas J y K para cada flip-flop. El control siguiente es necesario:

- Si un flip-flop en el estado 0 va a permanecer así, entonces $J = 0$ y $K = d$ (donde d significa que K puede ser igual a 0 o a 1).

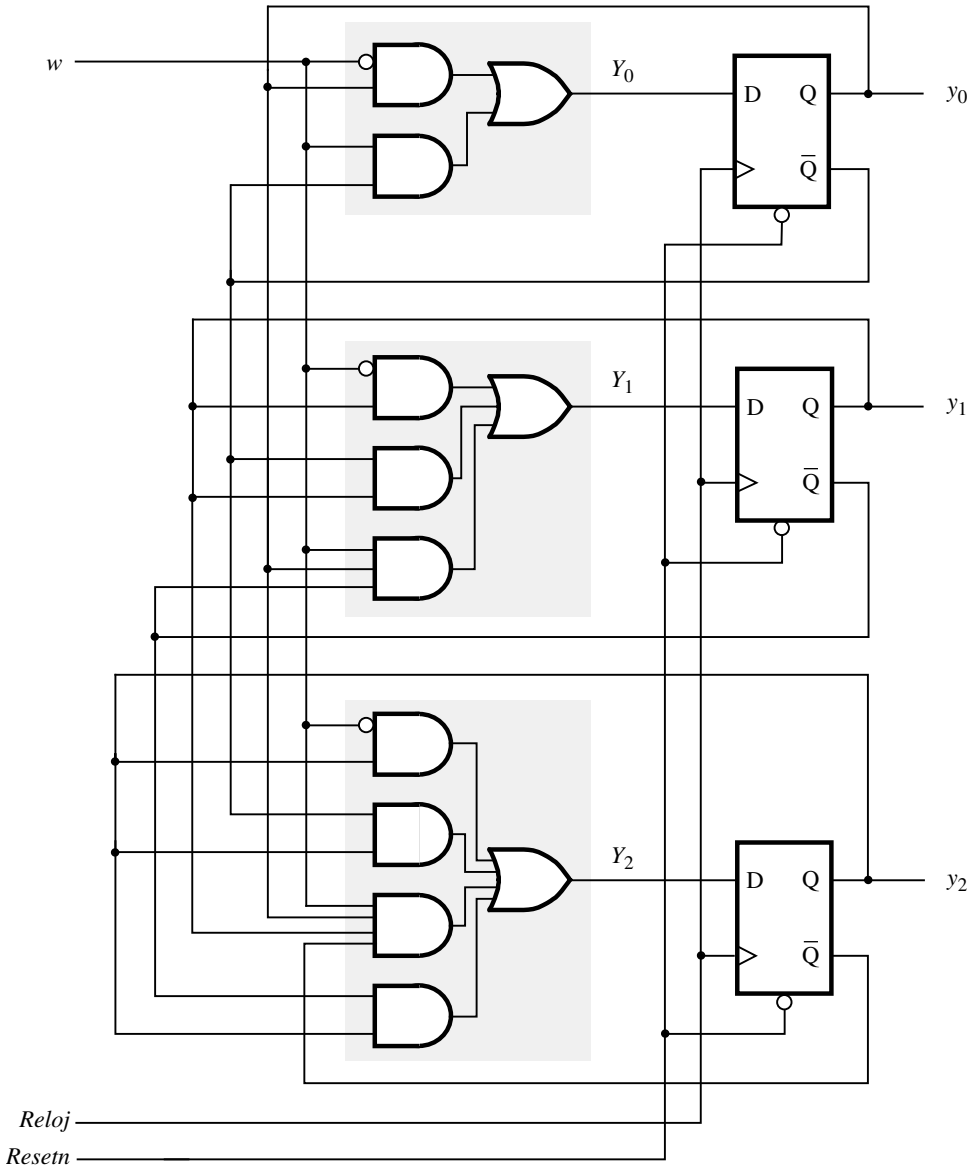


Figura 8.64 Diagrama de circuito para el contador implementado con flip-flops.

- Si un flip-flop en el estado 0 va a cambiar al estado 1, entonces $J = 1$ y $K = d$.
- Si un flip-flop en el estado 1 va a permanecer así, entonces $J = d$ y $K = 0$.
- Si un flip-flop en el estado 1 va a cambiar al estado 0, entonces $J = d$ y $K = 1$.

Siguiendo estas directrices podemos crear una tabla de verdad que especifique los valores requeridos de las entradas J y K para los tres flip-flops de nuestro diseño. En la figura 8.65 se muestra una versión modificada de la tabla de valores asignados de la figura 8.62, con las funciones de las entradas J y K incluidas. Para ver cómo se deriva esta tabla, considérese la primera fila, donde el estado presente es $y_2 y_1 y_0 = 000$. Si $w = 0$, entonces el estado siguiente también es $Y_2 Y_1 Y_0 = 000$. Por tanto, el valor actual en cada flip-flop es 0 y debe permanecer así. Esto implica el control de $J = 0$ y $K = d$ para los tres flip-flops. Al continuar con la primera fila, si $w = 1$, el estado siguiente será $Y_2 Y_1 Y_0 = 001$. De esta manera los flip-flops y_2 y y_1 seguirán estando en 0 y tendrán el control de $J = 0$ y $K = d$. Sin embargo, el flip-flop y_0 debe cambiar de 0 a 1, lo que se logra con $J = 1$ y $K = d$. El resto de la tabla se deriva del mismo modo al considerar cada estado presente $y_2 y_1 y_0$ y proporcionar las señales de control necesarias para alcanzar el nuevo estado $Y_2 Y_1 Y_0$.

Una tabla de asignación de estados es en esencia la tabla de estados en la que cada estado se codifica utilizando las variables de estado. Cuando los flip-flops D se usan para implementar una FSM, las entradas del estado siguiente en la tabla de asignación de estados corresponden directamente a las señales que deben aplicarse a las entradas D. Esto no es así si se emplea algún otro tipo de flip-flops. Una tabla que proporciona la información de los estados en forma de entradas de flip-flop que deben "excitarse" para ocasionar las transiciones a los estados siguientes se llama *tabla de excitación*. La tabla de excitación de la figura 8.65 indica de qué manera pueden utilizarse los flip-flops JK. En muchos libros el término *tabla de excitación* se utiliza incluso cuando los flip-flops D están involucrados, en cuyo caso es sinónimo de la tabla de asignación de estados.

Una vez que la tabla de la figura 8.65 se ha construido, proporciona una tabla de verdad con entradas y_2, y_1, y_0 y w, y salidas J_2, K_2, J_1, K_1, J_0 y K_0. Podemos entonces derivar las expresiones

	Estado presente $y_2y_1y_0$	Entradas de flip-flop								Conteo $z_2z_1z_0$
		$w = 0$				$w = 1$				
		$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	
A	000	000	$0d$	$0d$	$0d$	001	$0d$	$0d$	$1d$	000
B	001	001	$0d$	$0d$	$d0$	010	$0d$	$1d$	$d1$	001
C	010	010	$0d$	$d0$	$0d$	011	$0d$	$d0$	$1d$	010
D	011	011	$0d$	$d0$	$d0$	100	$1d$	$d1$	$d1$	011
E	100	100	$d0$	$0d$	$0d$	101	$d0$	$0d$	$1d$	100
F	101	101	$d0$	$0d$	$d0$	110	$d0$	$1d$	$d1$	101
G	110	110	$d0$	$d0$	$0d$	111	$d0$	$d0$	$1d$	110
H	111	111	$d0$	$d0$	$d0$	000	$d1$	$d1$	$d1$	111

Figura 8.65 Tabla de excitación para el contador con flip-flops JK.

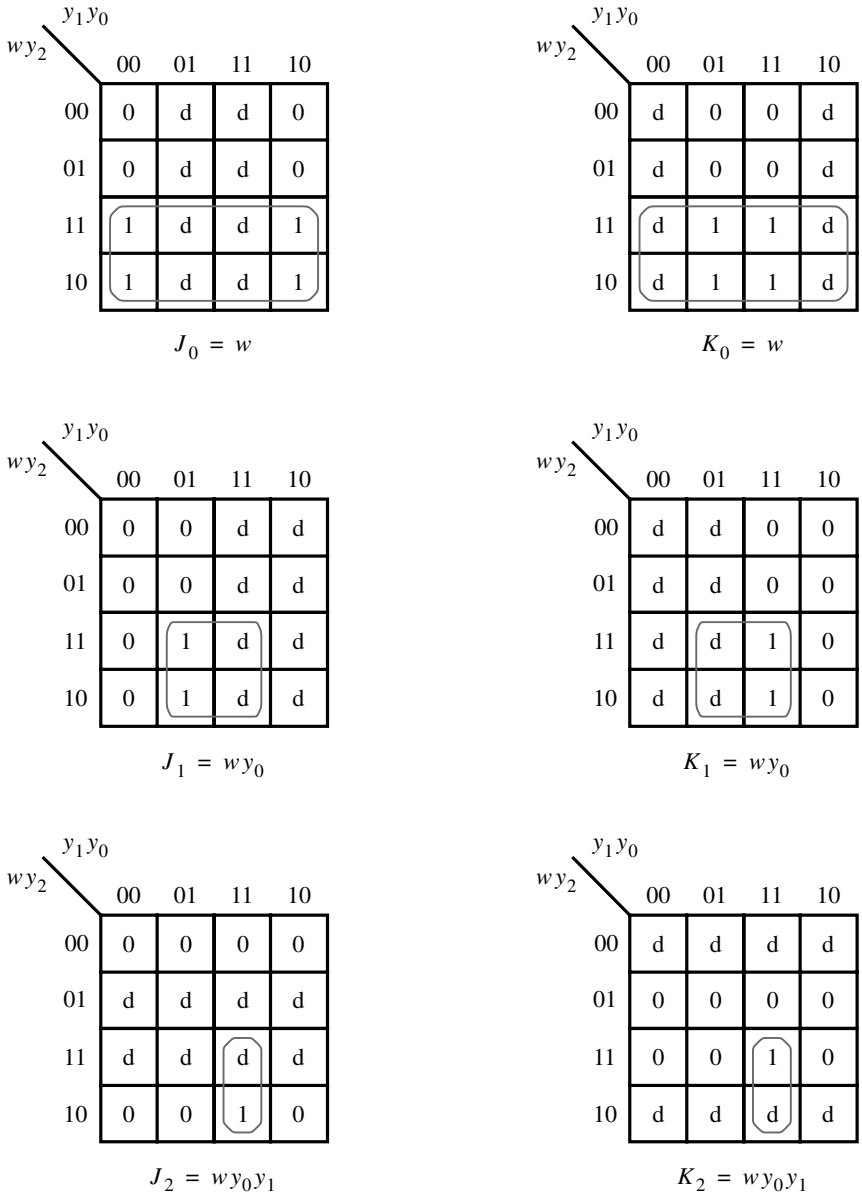


Figura 8.66 Mapas de Karnaugh para los flip-flops JK del contador.

para esas salidas, como se muestra en la figura 8.66. Las expresiones que resultan son

$$J_0 = K_0 = w$$
$$J_1 = K_1 = wy_0$$
$$J_2 = K_2 = wy_0y_1$$

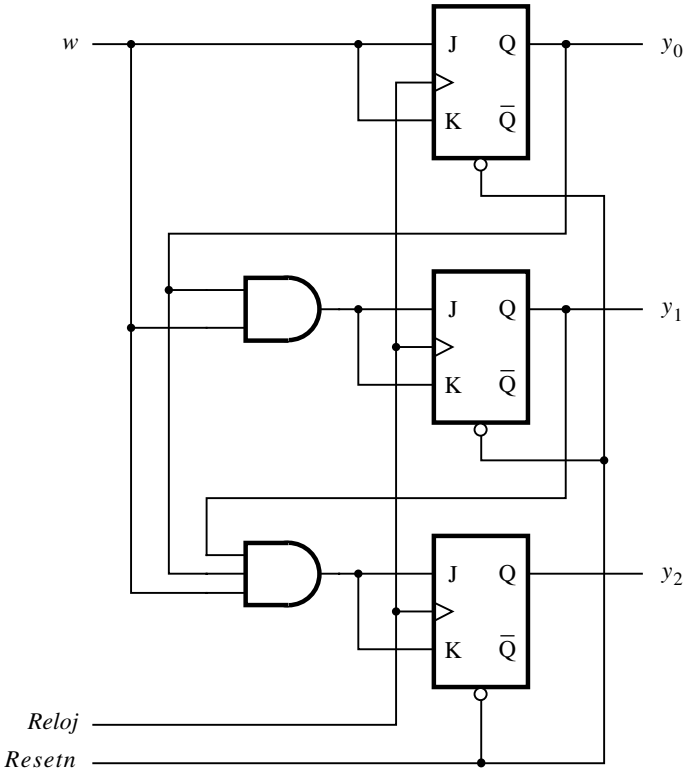


Figura 8.67 Diagrama de circuito que utiliza flip-flops JK.

Esto lleva al circuito mostrado en la figura 8.67. Es evidente que este diseño puede ampliarse fácilmente para contadores más grandes. El patrón $J_n = K_n = wy_0y_1 \ldots y_{n-1}$ define el circuito para cada etapa del contador. Obsérvese que el tamaño de la compuerta AND que implementa el término producto $y_0y_1 \ldots y_{n-1}$ aumenta con las etapas sucesivas. Un circuito con una estructura más regular se obtiene factorizando los términos requeridos antes conforme se avanza por las etapas del contador. Esto da

$$\begin{aligned} J_2 = K_2 &= (wy_0)y_1 &&= J_1y_1 \\ J_n = K_n &= (wy_0 \cdots y_{n-2})y_{n-1} &&= J_{n-1}y_{n-1} \end{aligned}$$

Usando la forma factorizada, el circuito contador puede realizarse como se indica en la figura 8.68. En este circuito todos los pasos (excepto el primero) parecen iguales. Nótese que este circuito tiene la misma estructura que el de la figura 7.23, ya que al conectar juntas las entradas *J* y *K* de un flip-flop éste se convierte en un flip-flop T.

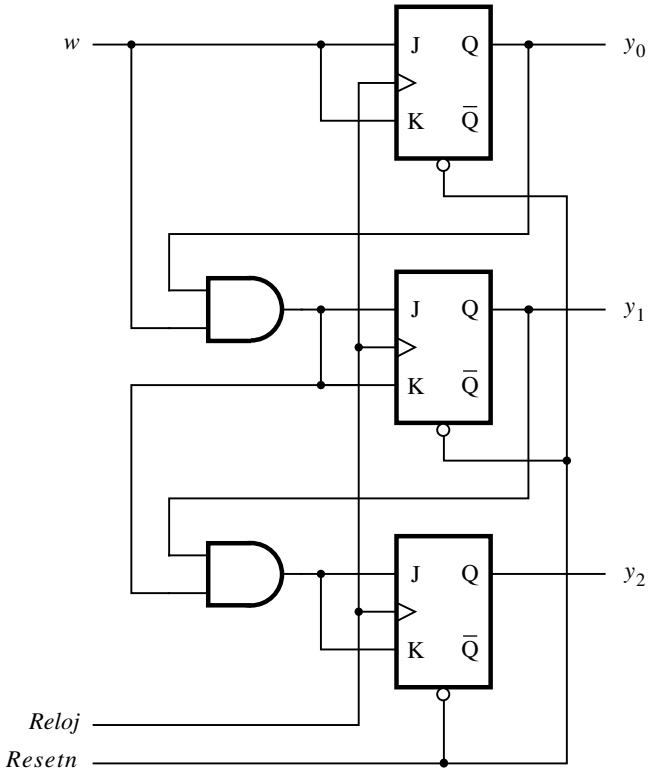


Figura 8.68 Implementación en forma factorizada del contador.

8.7.5 EJEMPLO. UN CONTADOR DIFERENTE

Ahora que hemos considerado el diseño de un contador ordinario, aplicaremos este conocimiento al diseño de un circuito tipo contador ligeramente distinto. Supóngase que deseamos derivar un contador de tres bits que cuenta los pulsos de una línea de entrada, *w*. Pero en vez de exhibir el conteo como 0, 1, 2, 3, 4, 5, 6, 7, 0, 1,..., este contador debe mostrar el conteo en la secuencia 0, 4, 2, 6, 1, 5, 3, 7, 0, 4 y así sucesivamente. El conteo estará representado directamente por los valores de los flip-flop mismos, sin emplear ninguna compuerta adicional. En concreto, Count = $Q_2Q_1Q_0$.

Como queremos contar los pulsos en la línea de entrada *w*, es lógico utilizar *w* como la entrada de reloj a los flip-flops. Por tanto, el circuito contador siempre debe estar habilitado y cambiar su estado cuando el siguiente pulso en la línea *w* aparezca. El contador buscado puede diseñarse de una manera sencilla por medio del enfoque de la FSM. En las figuras 8.69 y 8.70 se proporcionan la tabla de estado requerida y una asignación de estados adecuada. Al usar flip-flops D obtenemos las ecuaciones de estado siguiente

$$D_2 = Y_2 = \bar{y}_2$$
$$D_1 = Y_1 = y_1 \oplus y_2$$

Estado presente	Estado siguiente	Salida $z_2z_1z_0$
A	B	0 0 0
B	C	1 0 0
C	D	0 1 0
D	E	1 1 0
E	F	0 0 1
F	G	1 0 1
G	H	0 1 1
H	A	1 1 1

Figura 8.69 Tabla de estado para el ejemplo tipo contador.

Estado presente $y_2y_1y_0$	Estado siguiente $Y_2Y_1Y_0$	Salida $z_2z_1z_0$
0 0 0	1 0 0	0 0 0
1 0 0	0 1 0	1 0 0
0 1 0	1 1 0	0 1 0
1 1 0	0 0 1	1 1 0
0 0 1	1 0 1	0 0 1
1 0 1	0 1 1	1 0 1
0 1 1	1 1 1	0 1 1
1 1 1	0 0 0	1 1 1

Figura 8.70 Tabla de asignación de estados para la figura 8.69.

$$\begin{aligned} D_0 = Y_0 &= y_0\overline{y}_1 + y_0\overline{y}_2 + \overline{y}_0y_1y_2 \\ &= y_0(\overline{y}_1 + \overline{y}_2) + \overline{y}_0y_1y_2 \\ &= y_0 \oplus y_1y_2 \end{aligned}$$

Esto conduce al circuito de la figura 8.71.

El lector debe comparar este circuito con el contador ascendente normal de la figura 7.24. Considere las primeras tres etapas de ese contador, con la entrada *Enable* en 1 y *Clock* = w. Así, los dos circuitos son esencialmente los mismos, con una pequeña diferencia en el orden de los bits en el conteo. En la figura 7.24 el flip-flop superior corresponde al bit menos significativo del conteo, en tanto que en la figura 8.71 el flip-flop superior corresponde al bit más significativo del conteo. Esto no es una mera coincidencia. En la figura 8.70 el conteo requerido se define como $Count = y_2y_1y_0$. No obstante, si los patrones de bits que definen los estados se ven en orden inverso y se interpretan como números binarios, de modo que $Count = y_0y_1y_2$, entonces los

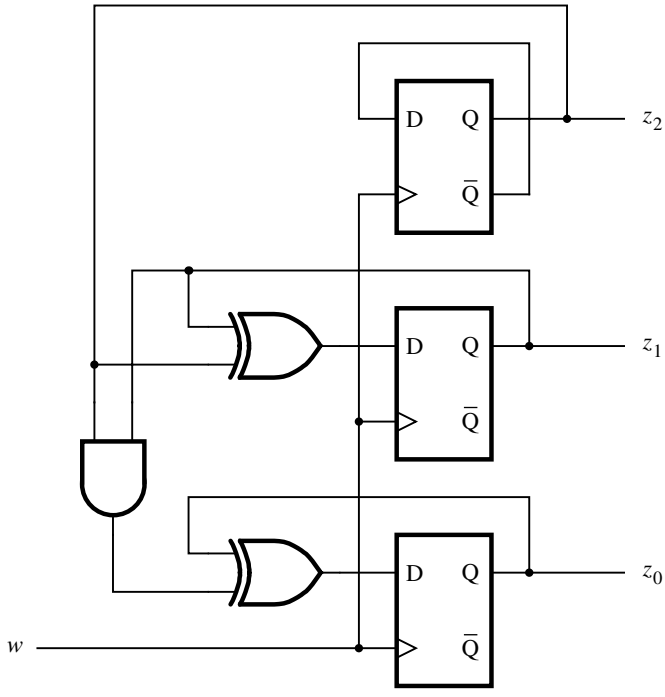


Figura 8.71 Circuito para la figura 8.70.

estados *A*, *B*, *C*,..., *H* tienen los valores 0, 1, 2,..., 7. Estos valores son los mismos que los valores asociados con el contador ascendente normal de tres bits.

8.8 FSM COMO UN CIRCUITO ÁRBITRO

En esta sección presentamos el diseño de una FSM un tanto más compleja que las de los ejemplos anteriores. El propósito de la máquina es controlar el acceso de varios dispositivos a un recurso compartido en un sistema. Sólo uno a la vez puede utilizar el recurso. Supóngase que todas las señales en el sistema pueden cambiar los valores únicamente en el flanco positivo de la señal de reloj. Cada dispositivo proporciona una entrada a la FSM, llamada *solicitud*, y la FSM produce una salida independiente para cada dispositivo, denominada *concesión*. Un dispositivo indica que necesita usar el recurso al validar su señal de solicitud. Siempre que el recurso compartido no está ya en uso, la FSM considera todas las solicitudes que estén activas. Con base en un esquema de prioridad, selecciona uno de los dispositivos que hacen una solicitud y valida su señal de concesión. Cuando el dispositivo termina de usar el recurso, invalida su señal de solicitud.

Supondremos que hay tres dispositivos en el sistema, llamados *dispositivo 1*, *dispositivo 2* y *dispositivo 3*. Es fácil ver cómo la FSM puede ampliarse para manejar más dispositivos. Las señales de solicitud se llaman r_1, r_2 y r_3, y las señales de concesión g_1, g_2 y g_3. Un nivel de prioridad se asigna a los dispositivos de tal manera que el dispositivo 1 tiene la mayor prioridad,

el dispositivo 2 la prioridad siguiente en importancia y el dispositivo 3 la menor prioridad. Por tanto, si se valida más de una señal de solicitud cuando la FSM asigna una concesión, ésta se da al dispositivo que tiene la solicitud con mayor prioridad.

Un diagrama de estado para la FSM buscada, diseñada como una máquina tipo Moore, se describe en la figura 8.72. Al principio, al inicializarla, la máquina se halla en el estado llamado *Idle* (inactivo). Ninguna señal de concesión se valida y el recurso compartido no está en uso. Hay otros tres estados, llamados *gnt*1, *gnt*2 y *gnt*3. Cada uno de ellos valida la señal para uno de los dispositivos.

La FSM permanece en el estado *Idle* siempre que todas las señales de solicitud sean 0. En el diagrama de estado la condición $r_1r_2r_3 = 000$ se indica por medio del arco etiquetado 000. Cuando una o más señales de solicitud se vuelven 1, la máquina pasa a uno de los estados de concesión, de acuerdo con el esquema de prioridad. Si r_1 se valida, entonces el dispositivo 1 recibirá la concesión porque tiene la prioridad más alta. Esto se indica con el arco etiquetado 1xx que conduce al estado *gnt*1, el cual establece $g_1 = 1$. El significado de 1xx es que la señal de solicitud r_1 es 1, y los valores de las señales r_2 y r_3 son irrelevantes debido al esquema de prioridad. Como antes, usamos el símbolo x para indicar que el valor de la variable correspondiente puede ser 0 o 1. La máquina permanece en el estado *gnt*1 mientras r_1 sea 1. Cuando $r_1 = 0$, el arco etiquetado 0xx produce en el siguiente flanco positivo del reloj un cambio de regreso al estado *Idle* y g_1 se invalida. Si se activan otras solicitudes en este momento, entonces la FSM cambiará a un nuevo estado de concesión después del siguiente flanco activo del reloj.

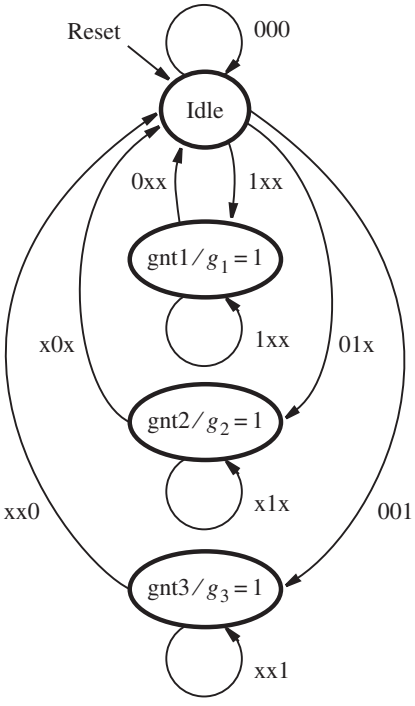


Figura 8.72 Diagrama de estado para el árbitro.

El arco que ocasiona un cambio en el estado *gnt*2 se llama 01x. Esta etiqueta se apega al esquema de prioridad porque representa la condición $r_2 = 1$, pero $r_1 = 0$. De forma similar, la condición para introducir el estado *gnt*3 se da como 001, lo cual indica que la única señal de solicitud validada es r_3.

El diagrama de estado se repite en la figura 8.73. La única diferencia entre este diagrama y el de la figura 8.72 es la forma en que se etiquetan los arcos. En la figura 8.73 se utiliza un esquema de etiquetado más simple, más intuitivo. Para la condición que lleva del estado *Idle* al estado *gnt*1, el arco se etiqueta r_1 en vez de 1xx. Esta etiqueta significa que si $r_1 = 1$, la FSM cambia al estado *gnt*1, independientemente de las otras entradas. El arco con la etiqueta $\bar{r}_1 r_2$ que lleva del estado *Idle* a *gnt*1 representa la condición $r_1 r_2 = 01$, mientras que el valor de r_3 es irrelevante. No existe un esquema estandarizado para etiquetar los arcos en los diagramas de estado. Algunos diseñadores prefieren el estilo de la figura 8.72; otros gustan más de un estilo parecido al de la figura 8.73.

En la figura 8.74 se presenta el código de VHDL para la máquina. Las tres señales de solicitud y de concesión se especifican como señales STD__LOGIC_ VECTOR de tres bits. La FSM se describe con una instrucción CASE en el estilo empleado para la figura 8.29. Como se muestra en la cláusula WHEN para el estado *Idle*, es fácil describir el esquema de prioridad requerido. Si la instrucción IF especifica que si $r_1 = 1$, entonces el estado siguiente para la

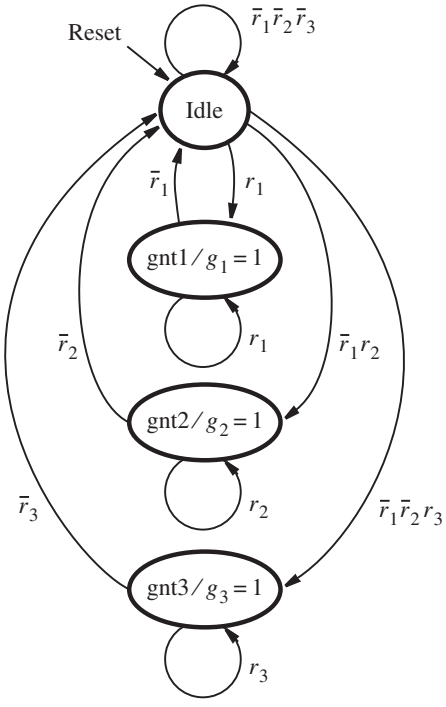


Figura 8.73 Estilo alternativo del diagrama de estado para el árbitro.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY arbiter IS
    PORT ( Clock, Resetn  : IN    STD_LOGIC ;
           r              : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           g              : OUT   STD_LOGIC_VECTOR(1 TO 3) ) ;
END arbiter ;

ARCHITECTURE Behavior OF arbiter IS
    TYPE State_type IS (Idle, gnt1, gnt2, gnt3) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= Idle ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN Idle =>
                    IF r(1) = '1' THEN y <= gnt1 ;
                    ELSIF r(2) = '1' THEN y <= gnt2 ;
                    ELSIF r(3) = '1' THEN y <= gnt3 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt1 =>
                    IF r(1) = '1' THEN y <= gnt1 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt2 =>
                    IF r(2) = '1' THEN y <= gnt2 ;
                    ELSE y <= Idle ;
                    END IF ;
                WHEN gnt3 =>
                    IF r(3) = '1' THEN y <= gnt3 ;
                    ELSE y <= Idle ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    g(1) <= '1' WHEN y = gnt1 ELSE '0' ;
    g(2) <= '1' WHEN y = gnt2 ELSE '0' ;
    g(3) <= '1' WHEN y = gnt3 ELSE '0' ;
END Behavior ;
```

Figura 8.74 Código de VHDL para el árbitro.

máquina es *gnt*1. Si r_1 no se valida, entonces la condición ELSIF se evalúa, la cual estipula que si $r_2 = 1$ entonces el estado siguiente será *gnt*2. Cada cláusula ELSIF sucesiva considera una señal de solicitud de la prioridad más baja sólo si todas las señales de solicitud de la prioridad más alta no se validan.

La cláusula WHEN para cada estado de concesión es sencilla. Para el estado *gnt*1 especifica que mientras $r_1 = 1$, el estado siguiente permanece en *gnt*1. Cuando $r_1 = 0$, el estado siguiente es *Idle*. Los otros estados de concesión tienen la misma estructura.

El código para las señales de concesión, g_1, g_2 y g_3, se proporciona al final. Establece g_1 en 1 cuando la máquina se halla en el estado *gnt*1; de otra forma, g_1 se establece en 0. De manera similar, cada una de las otras señales de concesión es 1 sólo en el estado de concesión apropiado.

En vez de las tres instrucciones de asignación condicionales utilizadas para g_1, g_2 y g_3, tal vez parezca razonable usar el proceso mostrado en la figura 8.75, el cual contiene una instrucción IF. Este código es incorrecto, pero el porqué no es obvio. Recuérdese del análisis de la figura 6.43 que cuando se usa una instrucción IF, si no hay una cláusula ELSE o un valor predeterminado para una señal, entonces esa señal conserva su valor cuando la condición IF no se cumple. Esto se denomina *memoria implícita*. En la figura 8.75 la señal g_1 se establece en 1 cuando la FSM entra por primera vez en el estado *gnt*1, y luego g_1 conservará el valor 1 sin importar a qué estado cambie la FSM. De igual modo, el código para g_2 y g_3 también es incorrecto. Si deseamos escribir el código que lleve una instrucción IF, entonces debemos estructurarlo como se muestra en la figura 8.76. Para cada señal de concesión se asigna un valor predeterminado de 0, con lo que se evita el problema de la memoria implícita.

8.8.1 IMPLEMENTACIÓN DEL CIRCUITO ÁRBITRO

Ahora consideraremos los efectos de la implementación del árbitro tanto en un CPLD como en un FPGA. Es probable que cualquier diferencia entre las dos implementaciones sea más pronunciada si la complejidad de la FSM es mayor. Por consiguiente, en vez de usar directamente el código de la figura 8.74, implementaremos una versión más grande del árbitro que controla ocho dispositivos. Las señales de solicitud se llaman r_1, r_2,..., r_8 y las de concesión g_1, g_2,..., g_8. Puesto que es fácil ver cómo se amplía el código de la figura 8.74 para permitir ocho dispositivos de solicitud, no lo mostraremos aquí.

```
        .
        .
        .
        PROCESS( y )
        BEGIN
            IF y = gnt1 THEN g(1) <= '1' ;
            ELSIF y = gnt2 THEN g(2) <= '1' ;
            ELSIF y = gnt3 THEN g(3) <= '1' ;
            END IF ;
        END PROCESS ;
END Behavior ;
```

Figura 8.75 Código de VHDL incorrecto para las señales de concesión.

```
        .
        .
        .
        PROCESS( y )
        BEGIN
            g(1) <= '0' ;
            g(2) <= '0' ;
            g(3) <= '0' ;
            IF y = gnt1 THEN g(1) <= '1' ;
            ELSIF y = gnt2 THEN g(2) <= '1' ;
            ELSIF y = gnt3 THEN g(3) <= '1' ;
            END IF ;
        END PROCESS ;
END Behavior ;
```

Figura 8.76 Código de VHDL correcto para las señales de concesión.

Implementación en un CPLD

Primero consideramos la implementación del árbitro en un CPLD. Para representar los nueve estados en la FSM, la herramienta de síntesis utiliza cuatro flip-flops, llamados y_4, y_3, y_2 y y_1. El código $y_4y_3y_2y_1 = 0000$ se asigna al estado de inicialización, *Idle*. Los otros estados se codifican como $gnt1 = 0001$, $gnt2 = 0010$, $gnt3 = 0100$, $gnt4 = 1000$, $gnt5 = 0011$, $gnt6 = 0101$, $gnt7 = 0110$ y $gnt8 = 1001$.

No es evidente por qué la herramienta de síntesis seleccionó esta asignación de estado en particular. La herramienta considera muchas asignaciones de estado y elige la que reduce al mínimo el costo del circuito final. Para la implementación del CPLD la herramienta de síntesis intenta elegir la asignación de estados que resulta en la menor cantidad de términos producto en el circuito final.

Para ver la complejidad del circuito debemos examinar las expresiones lógicas generadas tanto por las señales de concesión como por las entradas a los flip-flops de estado. La expresión de cada señal de concesión es un resultado directo de la codificación utilizada para el estado que produce la concesión. Por ejemplo, el estado *gnt*8 está codificado como 1001, lo que da por resultado $g_8 = y_4\,\bar{y}_3\,\bar{y}_2\,y_1$.

La alimentación lógica de los flip-flops de estados es más compleja. Por ejemplo, la expresión derivada por la herramienta para la entrada, Y_4, al flip-flop y_4 es

$$Y_4 = \bar{r}_1\bar{r}_2\bar{r}_3\bar{r}_5\bar{r}_6\bar{r}_7r_8\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4 + \bar{r}_1\bar{r}_2\bar{r}_3r_4\bar{y}_1\bar{y}_2\bar{y}_3 + r_8\bar{y}_1\bar{y}_2\bar{y}_3y_4 + r_4\bar{y}_1\bar{y}_2\bar{y}_3y_4$$

En la figura 8.77 se presenta una simulación de tiempo para la implementación en el CPLD. Por simplicidad sólo se muestran las señales de solicitud r_1, r_2 y r_8, junto con las señales de concesión g_1, g_2 y g_8. Después que la máquina se inicializa al principio de la simulación, las tres solicitudes r_1, r_2 y r_8 se validan. Aunque no se muestra en el diagrama de tiempo, todas las demás señales de solicitud se establecen en 0. La máquina primero se cambia al estado *gnt*1 y valida g_1. Después que r_1 se vuelve 0 la máquina cambia de nuevo al estado *Idle*. En el siguiente ciclo de reloj ocurre una transición al estado *gnt*2 y g_2 se valida. Después que r_2 se vuelve 0 la máquina cambia de nuevo al estado *Idle* y luego al estado *gnt*8 para validar g_8. Los resultados de la simulación indican que nuestro código de VHDL implementó de manera adecuada el esquema de prioridad requerido.

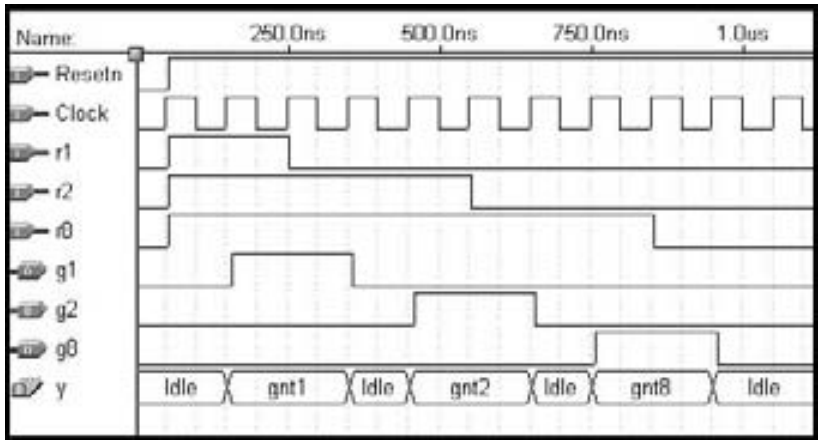


Figura 8.77 Resultado de la simulación para el circuito árbitro.

Una muestra más detallada de una parte de los resultados de la simulación aparece en la figura 8.78. Las formas de onda se acomodaron de modo que sólo las señales *Clock*, g_8 y y están visibles en el periodo durante el que g_8 se valida. Los resultados de la simulación muestran que se necesita un retraso de propagación (de alrededor de 7 ns) para que la señal g_8 se produzca después que la máquina cambia al estado *gnt*8. Este retraso corresponde al tiempo requerido para generar la función $g_8 = y_1 \overline{y}_2 \overline{y}_3 y_4$. En la sección 8.8.2 mostraremos que es posible mejorar la sincronización del circuito implementado de tal forma que una señal de concesión se produzca de inmediato cuando la máquina entra en el estado de concesión.

Implementación en un FPGA

Enseguida consideramos la implementación de la FSM árbitro en un chip FPGA (arreglo de compuertas programables por campo, *field-programmable gate array*). En vez de utilizar cuatro flip-flops para representar los nueve estados en la FSM, la implementación del FPGA generada por la herramienta de síntesis tiene nueve flip-flops de estado, llamados y_9, y_8,..., y_1. La asignación de estados es *Idle* = 000000000, *gnt*1 = 110000000, *gnt*2 = 101000000, *gnt*3 = 100100000, *gnt*4 = 100010000, *gnt*5 = 100001000, *gnt*6 = 100000100, *gnt*7 = 100000010 y *gnt*8 = 100000001. Esta asignación es muy parecida a la codificación de 1 activo. La única

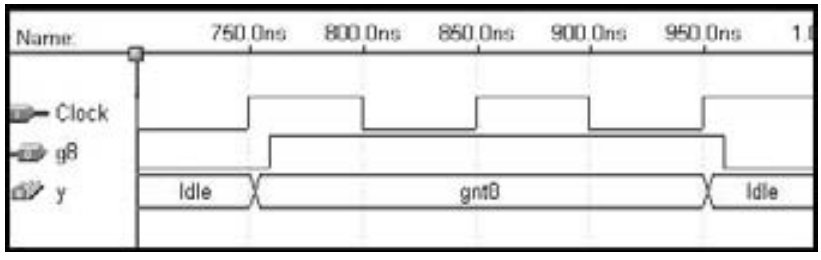


Figura 8.78 Retrasos de salida en el circuito árbitro.

diferencia estriba en que la salida del flip-flop en el extremo izquierdo, y_9, se complementa para proporcionar un mecanismo reset simple. Cuando todos los flip-flops se inicializan, definen el estado representado por todas las variables de estado que son 0, estado que es *Idle*.

En la sección 4.6 explicamos la cuestión de la carga de entrada limitada de las compuertas lógicas provistas en ciertos tipos de chips. Dijimos que en esos chips las funciones lógicas con un gran número de entradas deben descomponerse en funciones más pequeñas. Para una FSM, esto significa que si el circuito lógico que alimenta a cada flip-flop de estado tiene muchas entradas, entonces pueden requerirse varios niveles de compuertas. Ello aumenta los retrasos de propagación en el circuito y resulta en una velocidad de operación menor. Para la implementación anterior de la FSM árbitro en el CPLD, mostramos la expresión lógica para la entrada al flip-flop y_4. Si esa expresión se implementara en un FPGA que tiene tablas de consulta (LUT, *lookup tables*) de cuatro entradas, un circuito que tiene tres de las LUT conectadas en serie requeriría un total de ocho LUT.

Por el contrario, la opción de nueve variables de estado con la asignación de estados anterior da como resultado un circuito más simple. Como ejemplo, para la entrada al flip-flop y_8, la herramienta de síntesis produce $Y_8 = r_1 y_8 + r_1 \bar{y}_9$. Puesto que sólo tiene cuatro entradas, esta expresión puede realizarse en una tabla de consulta de cuatro entradas. Las otras ocho expresiones del estado siguiente también son relativamente simples. A fin de ver el efecto que tiene la asignación de estados en la velocidad de operación de la FSM comparamos las dos versiones del circuito implementado en un chip FPGA: uno que tiene nueve flip-flops de estado, como se mostró antes, y otro que tiene cuatro flip-flops con la asignación de estados dada con anterioridad para la implementación en el CPLD. Los resultados demostraron que cuando se utilizan nueve variables de estado, la FSM árbitro funciona correctamente hasta una velocidad de reloj máxima de 88.5 MHz, en tanto que cuando se usan cuatro variables de estado, la velocidad de reloj máxima es de sólo 54.1 MHz. Nótese que la velocidad de operación del circuito depende del chip objetivo específico y también puede variar con base en las opciones de síntesis seleccionadas en las herramientas CAD.

También debemos considerar la complejidad de la lógica necesaria para las señales de concesión. Estas señales son sencillas de generar cuando se utilizan nueve flip-flops. Cada señal de concesión es la salida de uno de los flip-flops. Por ejemplo, $g_8 = y_1$.

8.8.2 MINIMIZACIÓN DE LOS RETRASOS DE SALIDA PARA UNA FSM

En la figura 8.78 se muestra el retraso de propagación en que se incurre para producir las señales de concesión cuando el circuito árbitro se implementa en un CPLD. Una vez que el circuito cambia a un estado de concesión, la señal de concesión apropiada se valida después de un retraso de alrededor de 7 ns. El retraso es producido por el sistema de circuitos que genera la señal de concesión según los valores de los flip-flops de estado. Sin embargo, como mostramos en la implementación del FPGA, cuando se usa la codificación de 1 activo cada señal de concesión se proporciona como la salida de uno de los flip-flops de estado. Por consiguiente, no se precisa ningún circuito adicional para generar las señales de salida. En la figura 8.79 se muestra una simulación de tiempo cuando el circuito árbitro se implementa en un CPLD utilizando la codificación de 1 activo. El retraso es muy pequeño desde el momento en que el circuito entra en un estado de concesión hasta que se produce la señal de concesión. Se incurre en un retraso breve debido al tiempo requerido para la propagación a través del buffer que se halla entre la salida del flip-flop y el pin del chip CPLD, pero este retraso dura alrededor de 2 ns. Los diseñadores de circuitos secuenciales realizan este tipo de mejora del tiempo en la práctica, ya que las especificaciones de diseño a menudo requieren que las salidas se produzcan después de los retrasos más breves posibles.

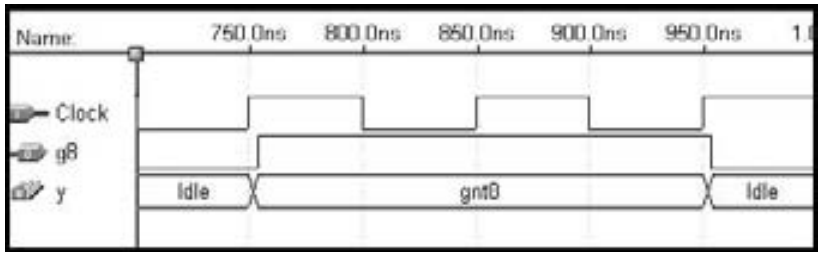


Figura 8.79 Retrasos de salida cuando se utiliza la codificación de 1 activo.

8.8.3 RESUMEN

Nuestra FSM árbitro es un circuito práctico útil en muchos tipos de sistemas. Un ejemplo es un sistema de cómputo en el que varios dispositivos se conectan por medio de un bus. Tal vez se requiera modificar un aspecto del árbitro para emplearlo en este tipo de sistema. Debido al esquema de prioridad, es posible que los dispositivos con una alta prioridad puedan impedir que un dispositivo de prioridad más baja reciba una señal de concesión durante un tiempo arbitrariamente prolongado. Esta condición con frecuencia se llama *inanición* del dispositivo de prioridad más baja. No es difícil modificar la FSM árbitro para atacar este inconveniente (véase problema 8.38).

8.9 ANÁLISIS DE LOS CIRCUITOS SECUENCIALES SÍNCRONOS

Además de saber cómo diseñar un circuito secuencial síncrono, el diseñador ha de ser capaz de analizar el comportamiento de un circuito existente. La tarea de análisis es mucho más simple que la de síntesis. En esta sección mostraremos cómo puede realizarse el análisis.

Para analizar un circuito, simplemente invertimos los pasos del proceso de síntesis. Las salidas de los flip-flops representan las variables del estado presente. Sus entradas determinan el estado siguiente en que entrará el circuito. A partir de esta información podemos construir la tabla de asignación de estados para el circuito. Esta tabla conduce a una tabla de estado y al diagrama de estado correspondiente al dar un nombre a cada estado. El tipo de flip-flops usados en el circuito es importante, como veremos en los ejemplos que siguen.

Ejemplo 8.8

FLIP-FLOPS D En la figura 8.80 se presenta una FSM que tiene dos flip-flops D. Sean y_1 y y_2 las variables del estado presente y Y_1 y Y_2 las del estado siguiente. Las expresiones de estado siguiente y de salida son

$$Y_1 = w\bar{y}_1 + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

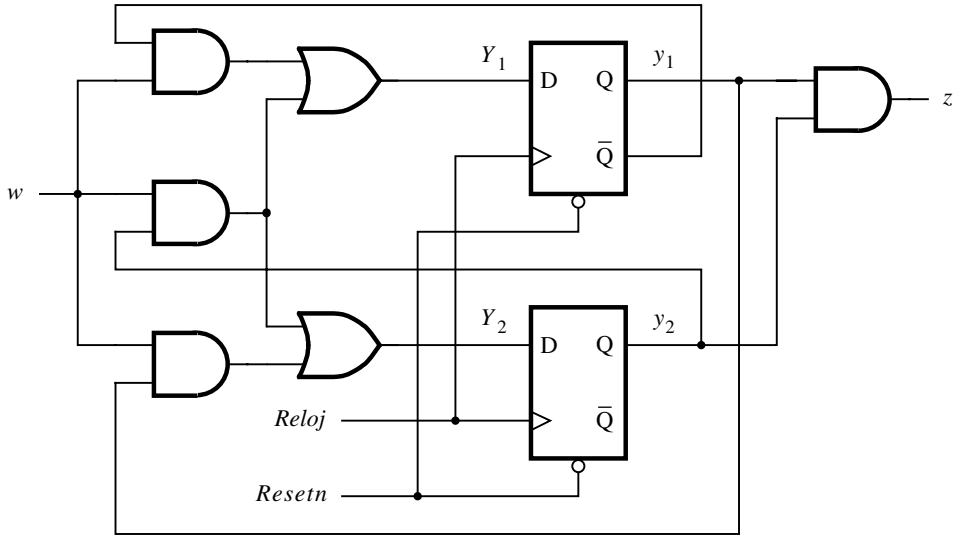


Figura 8.80 Circuito para el ejemplo 8.8.

Como hay dos flip-flops, la FSM tiene cuatro estados. Un buen punto de partida en el análisis es suponer un estado inicial de los flip-flops como $y_1 = y_2 = 0$. A partir de las expresiones para Y_1 y Y_2, podemos derivar la tabla de asignación de estados de la figura 8.81*a*. Por ejemplo, en la primera fila de la tabla $y_1 = y_2 = 0$. Entonces $w = 0$ hace que $Y_1 = Y_2 = 0$, y $w = 1$ ocasiona que $Y_1 = 1$ y $Y_2 = 0$. La salida para este estado es $z = 0$. Las otras filas se derivan de la misma forma. Etiquetar los estados como A, B, C y D produce la tabla de estado de la figura 8.81*b*. De esta tabla se advierte que después de la condición reset la FSM produce la salida $z = 1$ siempre que ocurren tres 1 consecutivos en la entrada w. Por consiguiente, la FSM actúa como un detector de secuencias para este patrón.

Ejemplo 8.9 **FLIP-FLOPS JK** Ahora considere el circuito de la figura 8.82, el cual tiene dos flip-flops JK. Las expresiones para las entradas a los flip-flops son

$$J_1 = w$$
$$K_1 = \overline{w} + \overline{y}_2$$
$$J_2 = wy_1$$
$$K_2 = \overline{w}$$

La salida está dada por $z = y_1 y_2$.

Con base en estas expresiones podemos derivar la tabla de excitación de la figura 8.83. La interpretación de las entradas de esta tabla nos permite construir una tabla de asignación de estados. Por ejemplo, considere $y_2 y_1 = 00$ y $w = 0$. Por tanto, como $J_2 = J_1 = 0$ y $K_2 = K_1 = 1$, los dos flip-flops permanecerán en el estado 0; en consecuencia, $Y_2 = Y_1 = 0$. Si $y_2 y_1 = 00$ y $w = 1$,

Estado presente y_2y_1	Estado siguiente $w=0$ Y_2Y_1	Estado siguiente $w=1$ Y_2Y_1	Salida z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

a) Tabla de asignación de estados

Estado presente	Estado siguiente $w=0$	Estado siguiente $w=1$	Salida z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

b) Tabla de estado

Figura 8.81 Tablas para el circuito de la figura 8.80.

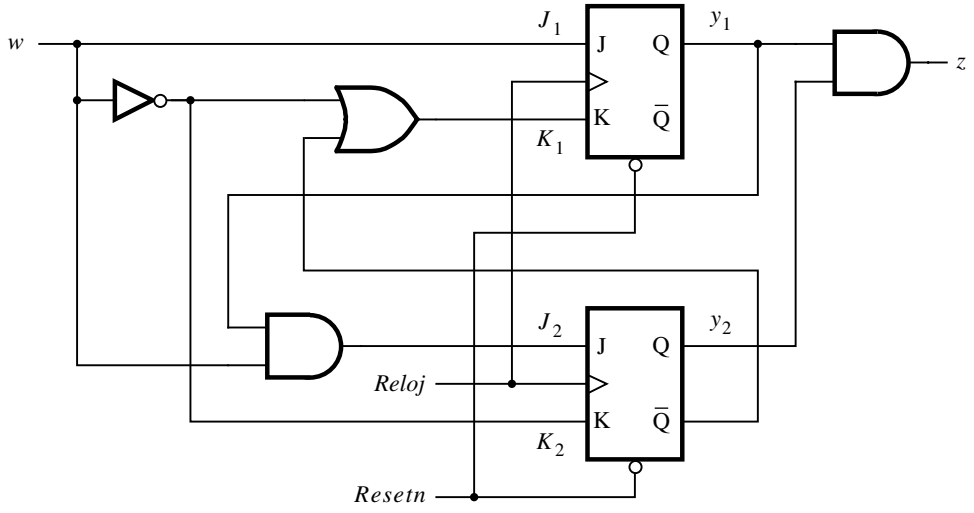


Figura 8.82 Circuito para el ejemplo 8.9.

Estado presente y_2y_1	Entradas de los flip-flops				Salida z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
0 0	0 1	0 1	0 0	1 1	0
0 1	0 1	0 1	1 0	1 1	0
1 0	0 1	0 1	0 0	1 0	0
1 1	0 1	0 1	1 0	1 0	1

Figura 8.83 Tabla de excitación para el circuito de la figura 8.82.

entonces $J_2 = K_2 = 0$ y $J_1 = K_1 = 1$, lo cual deja el flip-flop y_2 sin cambios y establece el flip-flop y_1 en 1; por consiguiente, $Y_2 = 0$ y $Y_1 = 1$. Si $y_2y_1 = 01$ y $w = 0$, entonces $J_2 = J_1 = 0$ y $K_2 = K_1 = 1$, lo que inicializa el flip-flop y_1 y ocasiona la transición al estado $y_2\ y_1 = 00$; por ende, $Y_2 = Y_1 = 0$. De manera similar, si $y_2y_1 = 01$ y $w = 1$, entonces $J_2 = 1$ y $K_2 = 0$ establecen y_2 en 1; en consecuencia, $Y_2 = 1$, mientras que $J_1 = K_1 = 1$ alterna el estado de y_1; de ahí que $Y_1 = 0$. Esto nos lleva al estado $y_2y_1 = 10$. Al completar este proceso encontramos que la tabla de asignación de estados resultante es la misma que la de la figura 8.81*a*. La conclusión es que los circuitos de las figuras 8.80 y 8.82 implementan la misma FSM.

Ejemplo 8.10 **FLIP-FLOPS COMBINADOS** No hay una razón por la que no podamos utilizar una combinación de flip-flops en un circuito. En la figura 8.84 se muestra un circuito con un flip-flop D y uno T. Las expresiones para este circuito son

$$D_1 = w(\overline{y}_1 + y_2)$$
$$T_2 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$z = y_1y_2$$

A partir de estas expresiones derivamos la tabla de excitación de la figura 8.85. Como se trata de un flip-flop T, y_2 cambia su estado sólo cuando $T_2 = 1$. Por tanto, si $y_2y_1 = 00$ y $w = 0$, entonces como $T_2 = D_1 = 0$ el estado del circuito no cambiará. Un ejemplo de cuando $T_2 = 1$ es en el caso que $y_2y_1 = 01$ y $w = 1$, lo cual hace que y_2 cambie a 1; $D_1 = 0$ hace que $y_1 = 0$, por lo que $Y_2 = 1$ y $Y_1 = 0$. Los otros casos en que $T_2 = 1$ ocurren cuando $w = 0$ y $y_2y_1 = 10$ u 11. En ambos casos $D_1 = 0$. Por consiguiente, el flip-flop T cambia su estado de 1 a 0, mientras que el D se borra, lo cual significa que el estado siguiente es $Y_2Y_1 = 00$. Al completar este análisis de nuevo obtenemos la tabla de asignación de estados de la figura 8.81*a*. Por tanto, este circuito es una implementación más de la FSM representada por la tabla de estado de la figura 8. 81*b*.

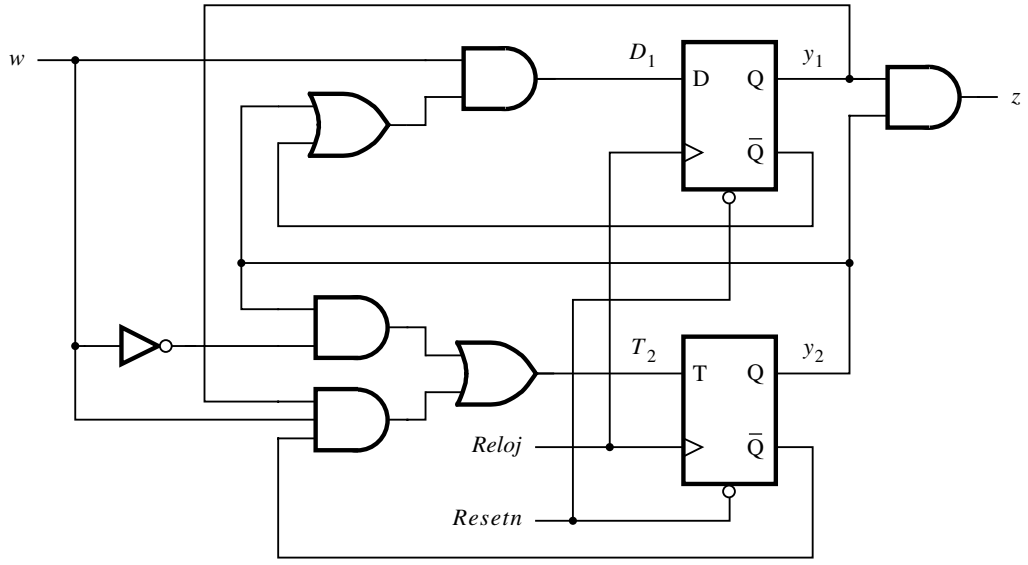


Figura 8.84 Circuito para el ejemplo 8.10.

Estado presente y_2y_1	Entradas de los flip-flops		Salida z
	$w = 0$	$w = 1$	
	T_2D_1	T_2D_1	
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	1 0	0 1	0
1 1	1 0	0 1	1

Figura 8.85 La tabla de excitación para el circuito de la figura 8.84.

8.10 CARTAS DE LA MÁQUINA ALGORÍTMICA DE ESTADOS (CARTAS ASM)

Los diagramas de estado y las tablas usadas en este capítulo resultan prácticos para describir el comportamiento de las FSM que sólo tienen algunas entradas y salidas. Para las máquinas más grandes los diseñadores a menudo utilizan una forma distinta de representación, llamada *carta de la máquina algorítmica de estados (carta ASM, algorithmic state machine)*.

Una carta ASM es un tipo de diagrama de flujo que puede emplearse para representar las transiciones de estado y las salidas generadas para una FSM. Los tres tipos de elementos utilizados en las cartas ASM se representan en la figura 8.86.

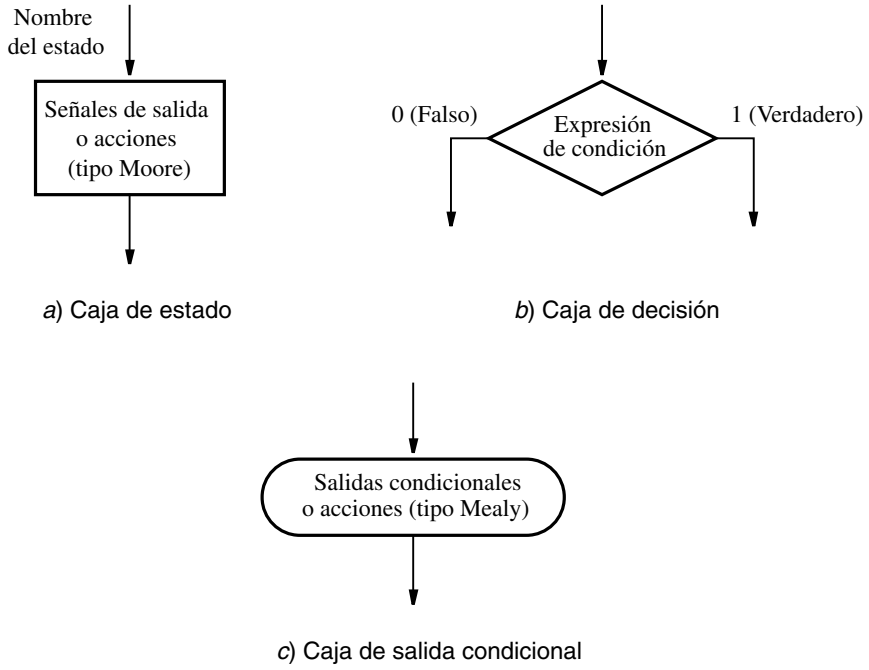


Figura 8.86 Elementos usados en las cartas ASM.

- **Caja de estado:** Un rectángulo representa un estado de la FSM. Equivale a un nodo en el diagrama de estado o a una fila de la tabla de estado. El nombre del estado se indica fuera de la caja, en la esquina superior izquierda. Las salidas tipo Moore se enumeran dentro de la caja. Éstas son las salidas que sólo dependen de los valores de las variables de estado que definen el estado; nos referiremos a ellas simplemente como *salidas Moore*. Es habitual escribir nada más el nombre de la señal que ha de validarse. Por tanto, basta escribir z en vez de $z = 1$ para indicar que la salida z debe tener el valor de 1. Además, tal vez sea útil indicar una acción que debe tomarse; por ejemplo, $Count \leftarrow Count + 1$ especifica que el contenido de un contador debe incrementarse en 1. Desde luego, esto es sólo un modo de decir que la señal de control que hace que el contador se incremente debe validarse. Utilizaremos esta manera de especificar las acciones en los sistemas más grandes que se estudian en el capítulo 10.
- **Caja de decisión:** Un diamante indica que la expresión de condición establecida se va a probar y la trayectoria de salida se elegirá en consecuencia. La expresión de condición consta de una o más entradas a la FSM. Por ejemplo, w indica que la decisión se basa en el valor de la entrada w, mientras que $w_1 \cdot w_2$ indica que la trayectoria de verdadero se toma si $w_1 = w_2 = 1$ y la de falso en caso contrario.
- **Caja de salida condicional:** Un óvalo indica las señales de salida que son tipo Mealy. Estas salidas dependen de los valores de las variables de estado y las entradas de la FSM; nos referiremos a ellas simplemente como *salidas Mealy*. La condición que determina si dichas salidas se generan se especifica en una caja de decisión.

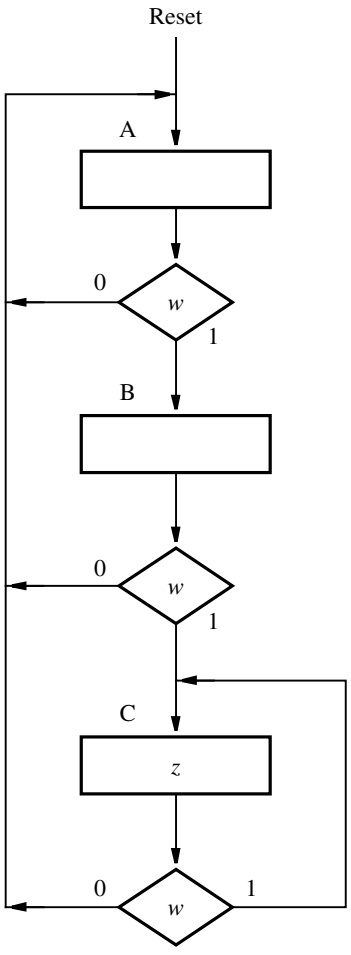


Figura 8.87 Carta ASM para la FSM de la figura 8.3.

En la figura 8.87 se proporciona la carta ASM que representa la FSM de la figura 8.3. Las transiciones entre las cajas de estado dependen de las decisiones tomadas al probar el valor de la variable de entrada *w.* En cada caso, si $w = 0$, la trayectoria de salida de una caja de decisión conduce al estado *A*. Si $w = 1$, entonces ocurre una transición de *A* a *B* o de *B* a *C*. Si $w = 1$ en el estado C, entonces la FSM permanece en ese estado. En la tabla se especifica una salida Moore *z*, la cual se valida sólo en el estado *C*, como se indica en la caja de estado. En los estados *A* y *B*, el valor de *z* es 0 (no está validado), lo que se da a entender al dejar las cajas de estado correspondientes en blanco.

En la figura 8.88 se brinda un ejemplo con salidas Mealy. En esta figura se representa la FSM de la figura 8.23. La salida, *z*, es igual a 1 cuando la máquina se halla en el estado *B* y $w = 1$. Esto se indica utilizando la caja de salida condicional. En todos los demás casos el valor de *z* es 0, lo que queda implícito al no especificar *z* como una salida del estado *B* para $w = 0$ y el estado A para *w* igual a 0 o 1.

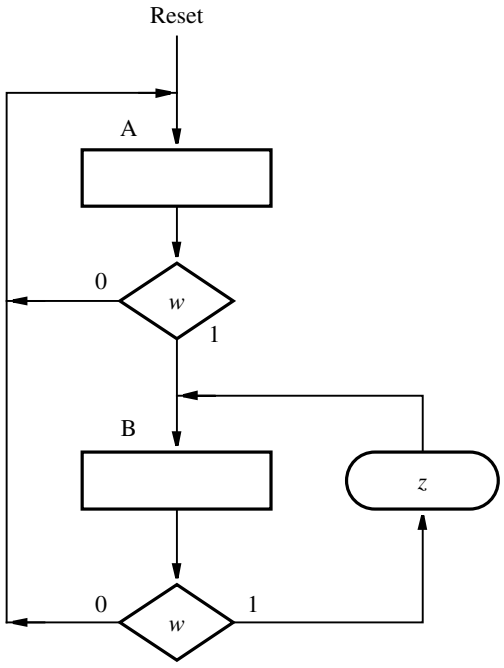


Figura 8.88 Carta ASM para la FSM de la figura 8.23.

En la figura 8.89 se presenta la carta ASM para la FSM árbitro de la figura 8.73. La caja de decisión dibujada debajo de la caja de estado para *Idle* especifica que si $r_1 = 1$, entonces la FSM cambia al estado *gnt*1. En tal estado la FSM valida la señal de salida g_1. La caja de decisión a la derecha de la caja de estado para *gnt*1 especifica que mientras $r_1 = 1$, la máquina permanece en el estado *gnt*1, y cuando $r_1 = 0$, cambia al estado *Idle*. La caja de decisión etiquetada r_2 que se dibujó debajo de la caja de estado para *Idle* especifica que si $r_2 = 1$, entonces la FSM cambia al estado *gnt*2. Esta caja de decisión sólo puede alcanzarse después de verificar el valor de r_1 y seguir la flecha que corresponde a $r_1 = 0$. De forma similar, la caja de decisión etiquetada r_3 puede alcanzarse sólo si tanto r_1 como r_2 tienen el valor 0. Por consiguiente, la carta ASM describe el esquema de prioridad requerido para el árbitro.

Las cartas ASM son similares a los diagramas de flujo tradicionales. Pero a diferencia de éstos la carta ASM incluye información de sincronización porque especifica de manera implícita que la FSM cambia (fluye) de un estado a otro sólo después de cada flanco activo del reloj. Los ejemplos de cartas ASM presentados aquí son muy simples. Los hemos utilizado para exponer la terminología de la carta ASM dando ejemplos de cajas de estado, de decisión y de salida condicional. Otro término que a veces se aplica a las cartas ASM es *bloque ASM*, que se refiere a una sola caja de estado y cualesquiera caja de decisión y de salida condicional con las que la caja de estado pueda estar conectada. Las cartas ASM pueden usarse para describir circuitos complejos que incluyen una o más máquinas de estado finito y otros sistemas de circuitos como registros, registros de corrimiento, contadores, sumadores y multiplicadores. Emplearemos las cartas ASM como ayuda para diseñar circuitos más complejos en el capítulo 10.

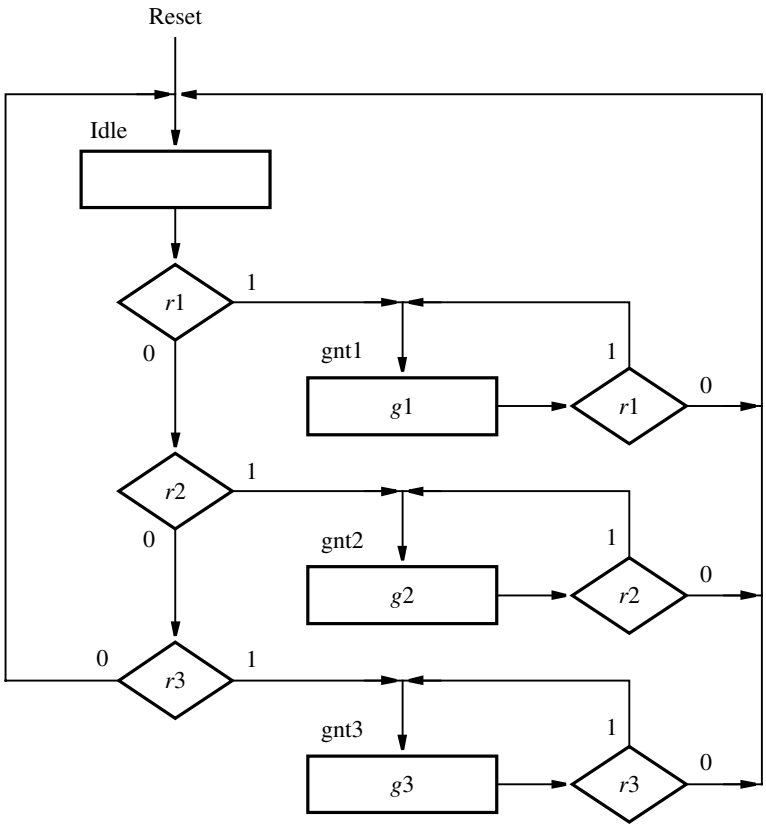


Figura 8.89 Carta ASM para la FSM árbitro de la figura 8.73.

8.11 MODELO FORMAL PARA CIRCUITOS SECUENCIALES

En este capítulo hemos presentado los circuitos secuenciales síncronos utilizando un método más bien informal porque ésta es la manera más sencilla de captar los conceptos esenciales del diseño de tales circuitos. Los mismos temas también pueden presentarse de un modo más formal, que es el estilo adoptado en muchos libros que destacan aspectos de la teoría de la conmutación en vez del diseño con herramientas CAD. Un modelo formal a menudo proporciona una especificación concisa difícil de igualar en una presentación más descriptiva. En esta sección describiremos un modelo formal que representa una clase general de circuitos secuenciales, incluidos los del tipo síncrono.

En la figura 8.90 se representa un circuito secuencial general. Tiene entradas $W = \{w_1, w_2,..., w_n\}$, salildas $Z = \{z_1, z_2,..., z_m\}$, variables de estado presente $y = \{y_1, y_2,..., y_k\}$ y variables de estado siguiente $Y = \{Y_1, Y_2,..., Y_k\}$. Puede tener hasta 2^k estados, $S = \{S_1, S_2,..., S_{2^k}\}$. Existen elementos de retraso en las trayectorias de retroalimentación para las variables de estado que aseguran que y tomará los valores de Y después de un retraso de tiempo Δ. En el caso de los circuitos secuenciales síncronos, los elementos del retraso son flip-flops, que cambian su

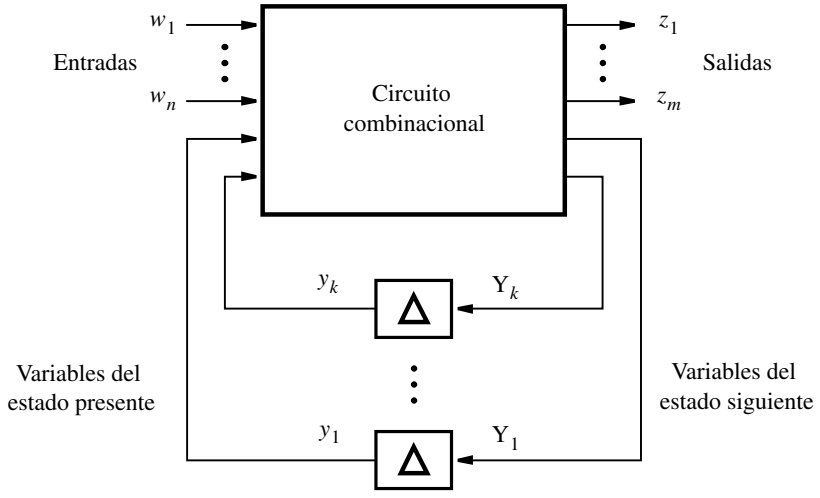


Figura 8.90 El modelo general para un circuito secuencial.

estado en el flanco activo de una señal de reloj. Por tanto, el retraso Δ está determinado por el periodo del reloj. Éste debe ser lo suficientemente largo para permitir el retraso de propagación en el circuito combinacional, además de los parámetros de preparación y espera de los flip-flops.

Con base en el modelo de la figura 8.90, un circuito secuencial síncrono, M, puede definirse formalmente como una quíntupla

$$M = (W, Z, S, \varphi, \lambda)$$

donde

- W, Z y S son conjuntos finitos, no vacíos, de entradas, salidas y estados, respectivamente.
- φ es la función de transición de estado, de modo que $S(t + 1) = \varphi[W(t), S(t)]$.
- λ es la función de salida, de modo que $\lambda(t) = \lambda[S(t)]$ para el modelo Moore y $\lambda(t) = \lambda[W(t), S(t)]$ para el modelo Mealy.

Esta definición supone que el tiempo entre t y $t + 1$ es un ciclo de reloj.

En el capítulo siguiente veremos que el retraso Δ no necesita estar controlado por un reloj. En los circuitos secuenciales asíncronos los retardos se deben únicamente a los retrasos de propagación a través de varias compuertas.

8.12 COMENTARIOS FINALES

La existencia de lazos cerrados y retrasos en un circuito secuencial conduce a un comportamiento caracterizado por una serie de estados en que el circuito puede entrar. Los valores presentes de las entradas no son el único factor determinante en este comportamiento, ya que la combinacion dada de las entradas puede causar que el circuito se comporte de una manera diferente en distintos estados.

Los retrasos de propagación a lo largo un circuito secuencial deben tomarse en cuenta. Las técnicas de diseño presentadas en este capítulo se basan en la suposición de que todos los cambios en el circuito se disparan por el flanco activo de una señal de reloj. Estos circuitos trabajan correctamente sólo si todas las señales internas son estables cuando la señal de reloj llega. Por tanto, el periodo del reloj debe ser mayor que el retraso de propagación más largo en el circuito.

En los diseños prácticos se usan mucho los circuitos secuenciales síncronos. Están soportados por las herramientas CAD de uso común. Todos los libros de texto de diseño de circuitos lógicos dedican un espacio considerable a los circuitos secuenciales síncronos. Algunas de las obras de consulta más notables son [1-14].

En el próximo capítulo presentaremos una clase diferente de circuitos secuenciales, la cual no utiliza flip-flops para representar los estados del circuito ni pulsos de reloj para disparar cambios en los estados.

8.13 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas comunes que el lector puede encontrar y se muestra cómo resolverlos.

Ejemplo 8.11

Problema: Se debe diseñar una FSM que tiene una entrada w y una salida z. La máquina es un detector de secuencias que produce $z = 1$ cuando los dos valores previos de w eran 00 o 11; de lo contrario, $z = 0$.

Solución: En la sección 8. 1 se presenta el diseño de un detector de secuencias que detecta la ocurrencia de 1 consecutivos. Con el mismo enfoque, la FSM deseada puede especificarse utilizando el diagrama de estado de la figura 8.91. El estado C indica la ocurrencia de dos o más 0, y el estado E indica dos o más 1. La tabla de estado correspondiente se muestra en la figura 8.92.

Podemos tratar de reducir el número de estados usando el procedimiento de minimización por particionamiento expuesto en la sección 8.6, el cual proporciona las particiones siguientes

$$P_1 = (ABCDE)$$
$$P_2 = (ABD)(CE)$$
$$P_3 = (A)(B)(C)(D)(E)$$

Como los cinco estados son necesarios, debemos usar tres flip-flops.

Una asignación de estados sencilla conduce a la tabla de asignación de estados de la figura 8.93. Los códigos $y_3 y_2 y_1 = 101, 110, 111$ pueden tratarse como condiciones no-importa. Por tanto, las expresiones de estado siguiente son

$$Y_1 = w\bar{y}_1\bar{y}_3 + w\bar{y}_2\bar{y}_3 + \bar{w}y_1y_2 + \overline{wy}_1\bar{y}_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2 + w\bar{y}_2\bar{y}_3$$
$$Y_3 = wy_3 + wy_1y_2$$

La expresión de salida es

$$z = y_3 + \bar{y}_1y_2$$

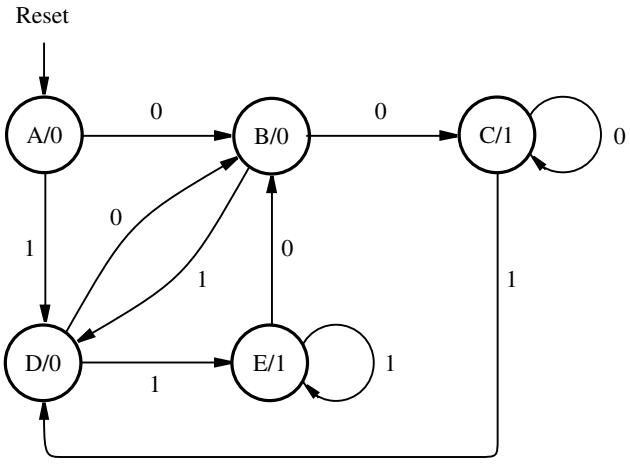


Figura 8.91 Diagrama de estado para el ejemplo 8.11.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	B	D	0
B	C	D	0
C	C	D	1
D	B	E	0
E	B	E	1

Figura 8.92 Tabla de estado para la FSM de la figura 8.91.

	Estado presente $y_3y_2y_1$	Estado siguiente		Salida
		$w = 0$	$w = 1$	z
		$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	
A	000	001	011	0
B	001	010	011	0
C	010	010	011	1
D	011	001	100	0
E	100	001	100	1

Figura 8.93 Tabla de asignación de estados para la FSM de la figura 8.92.

	Estado presente $y_3y_2y_1$	Estado siguiente $w = 0$ $Y_3Y_2Y_1$	Estado siguiente $w = 1$ $Y_3Y_2Y_1$	Salida z
A	000	100	110	0
B	100	101	110	0
C	101	101	110	1
D	110	100	111	0
E	111	100	111	1

Figura 8.94 Asignación de estados mejorada para la FSM de la figura 8.92.

Estas expresiones parecen innecesariamente complejas, lo que sugiere que podríamos hallar una asignación de estados mejor. Observe que el estado *A* se alcanza sólo cuando la máquina se inicializa por medio de la entrada *Reset*. Así que tal vez sea recomendable asignar los cuatro códigos en los que $y_3 = 1$ a los estados *B*, *C*, *D* y *E*. El resultado es la tabla de asignación de estados de la figura 8.94. A partir de ella, las expresiones de estado siguiente y de salida son

$$\begin{aligned} Y_1 &= wy_2 + \overline{w}y_3\overline{y}_2 \\ Y_2 &= w \\ Y_3 &= 1 \\ z &= y_1 \end{aligned}$$

que es una mucho mejor solución.

Ejemplo 8.12

Problema: Implemente el detector de secuencias del ejemplo 8.11 utilizando dos FSM. Una detecta la ocurrencia de 1 consecutivos, mientras que la otra detecta la de 0 consecutivos.

Solución: Una buena realización de la FSM que detecta 1 consecutivos se da en las figuras 8.16 y 8.17. Las expresiones de estado siguiente y de salida son

$$\begin{aligned} Y_1 &= w \\ Y_2 &= wy_1 \\ z_{ones} &= y_2 \end{aligned}$$

Una FSM similar que detecta 0 consecutivos se define en la figura 8.95. Sus expresiones son

$$\begin{aligned} Y_3 &= \overline{w} \\ Y_4 &= \overline{w}y_3 \\ z_{zeros} &= y_4 \end{aligned}$$

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z_{zeros}
D	E	D	0
E	F	D	0
F	F	D	1

a) Tabla de estado

	Estado presente	Estado siguiente		Salida
		$w = 0$	$w = 1$	z_{zeros}
	y_4y_3	Y_4Y_3	Y_4Y_3	
D	00	01	00	0
E	01	11	00	0
F	11	11	00	1
	10	dd	dd	d

b) Tabla de asignación de estados

Figura 8.95 FSM que detecta una secuencia de dos ceros.

La salida del circuito combinado es

$$z = z_{ones} + z_{zeros}$$

Ejemplo 8.13 **Problema:** Derive una FSM tipo Mealy que pueda actuar como el detector de secuencias descrito en el ejemplo 8.11.

Solución: Un diagrama de estado para la FSM buscada se presenta en la figura 8.96. La tabla de estado correspondiente se muestra en la figura 8.97. Se necesitan dos flip-flops para implementar esta FSM. En la figura 8.98 aparece una tabla de asignación de estados, la cual conduce a las expresiones de estado siguiente y de salida

$$Y_1 = 1$$
$$Y_2 = w$$
$$z = \overline{w}y_1\overline{y}_2 + wy_2$$

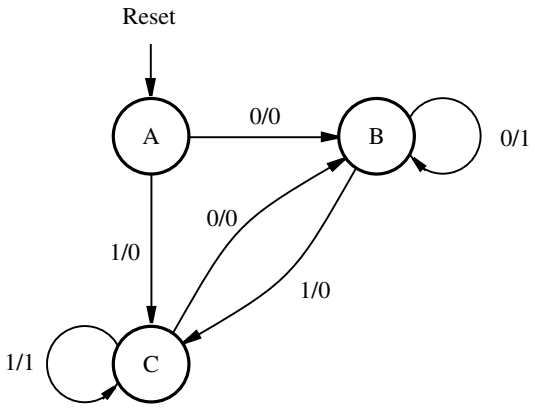


Figura 8.96 Diagrama de estado para el ejemplo 8.13.

Estado presente	Estado siguiente		Salida z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	B	C	1	0
C	B	C	0	1

Figura 8.97 Tabla de estado para la FSM de la figura 8.96.

	Estado presente	Estado siguiente		Salida	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y_2y_1	Y_2Y_1	Y_2Y_1	z	z
A	00	01	11	0	0
B	01	01	11	1	0
C	11	01	11	0	1

Figura 8.98 Tabla de asignación de estados para la FSM de la figura 8.97.

	Estado presente $y_3y_2y_1$	Entradas de flip-flop								Salida z
		$w = 0$				$w = 1$				
		$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	
A	000	100	1d	0d	0d	110	1d	1d	0d	0
B	100	101	d0	0d	1d	110	d0	1d	0d	0
C	101	101	d0	0d	d0	110	d0	1d	d1	1
D	110	100	d0	d1	0d	111	d0	d0	1d	0
E	111	100	d0	d1	d1	111	d0	d0	d0	1

Figura 8.99 Tabla de excitación para la FSM de la figura 8.94 con flip-flops JK.

Ejemplo 8.14 **Problema:** Implemente la FSM de la figura 8.94 utilizando flip-flops JK.

Solución: En la figura 8.99 se muestra la tabla de excitación. Ésta da como resultado las siguientes expresiones de estado siguiente y salida

$$
\begin{aligned}
J_1 &= wy_2 + \overline{w}y_3\overline{y}_2 \\
K_1 &= \overline{w}y_2 + wy_1\overline{y}_2 \\
J_2 &= w \\
K_2 &= \overline{w} \\
J_3 &= 1 \\
K_3 &= 0 \\
z &= y_1
\end{aligned}
$$

Ejemplo 8.15 **Problema:** Escriba código de VHDL para implementar la FSM de la figura 8.91.

Solución: Usando el estilo de código dado en la figura 8.29, la FSM requerida puede especificarse como se muestra en la figura 8.100.

Ejemplo 8.16 **Problema:** Escriba código de VHDL para implementar la FSM de la figura 8.96.

Solución: Utilizando el estilo de código dado en la figura 8.36, la FSM tipo Mealy puede especificarse como se muestra en la figura 8.101.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY sequence IS
    PORT ( Clock, Resetn, w     : IN     STD_LOGIC ;
           z                    : OUT  STD_LOGIC ) ;
END sequence ;

ARCHITECTURE Behavior OF sequence IS
    TYPE State_type IS (A, B, C, D, E) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= D ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= C ;
                    ELSE y <= D ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN y <= C ;
                    ELSE y <= D ;
                    END IF ;
                WHEN D =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= E ;
                    END IF ;
                WHEN E =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= E ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN (y = C OR y = E) ELSE '0' ;
END Behavior ;
```

Figura 8.100 Código de VHDL para la FSM de la figura 8.91.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY seqmealy IS
    PORT ( Clock, Resetn, w     : IN    STD_LOGIC ;
           z                    : OUT   STD_LOGIC ) ;
END seqmealy ;

ARCHITECTURE Behavior OF seqmealy IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN y <= B ;
                    ELSE y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
                WHEN A =>
                    z <= '0' ;
                WHEN B =>
                    z <= NOT w ;
                WHEN C =>
                    z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura 8.101 Código de VHDL para la FSM de la figura 8.96.

Ejemplo 8.17

Problema: En los sistemas de cómputo con frecuencia es deseable transmitir datos en forma serial, concretamente, un bit a la vez, para ahorrar en el costo de los cables de interconexión. Esto significa que los datos paralelos en un extremo deben transmitirse serialmente, y en el otro extremo los datos seriales recibidos han de convertirse de nuevo en paralelos. Suponga que queremos transmitir caracteres ASCII de esta manera. Como se explicó en la sección 5.8, el código ASCII estándar utiliza siete bits para definir cada carácter. Por lo general, un carácter ocupa un byte, en cuyo caso el octavo bit puede ya sea establecerse en 0 o usarse para indicar la paridad de los otros bits a fin de asegurar una transmisión más confiable.

La conversión de paralelo a serial puede realizarse por medio de un registro de corrimiento. Suponga que un circuito acepta datos paralelos, $B = b_7, b_6,..., b_0$, que representan caracteres ASCII. Suponga también que el bit b_7 se establece en 0. Se asume que el circuito genera un bit de paridad, p, y lo envía en vez de b_7 como parte de la transferencia serial. En la figura 8.102 se muestra un circuito posible. Una FSM se utiliza para generar el bit de paridad, el cual se incluye en el flujo de salida mediante un multiplexor. Se emplea un contador de tres bits para determinar cuándo se transmite el bit p, lo que ocurre cuando el conteo llega a 7. Diseñe la FSM buscada.

Solución: A medida que los bits se mueven hacia fuera del registro de corrimiento, la FSM los examina y lleva la cuenta de si ha sido un número par o impar de unos (1). Establece p en 1 si la paridad es impar. Por consiguiente, la FSM debe tener dos estados. En la figura 8.103 se presenta la tabla de estado, la tabla de asignación de estados y el circuito resultante. La expresión de estado siguiente es

$$Y = \overline{w}y + w\overline{y}$$

La salida p es simplemente igual a y.

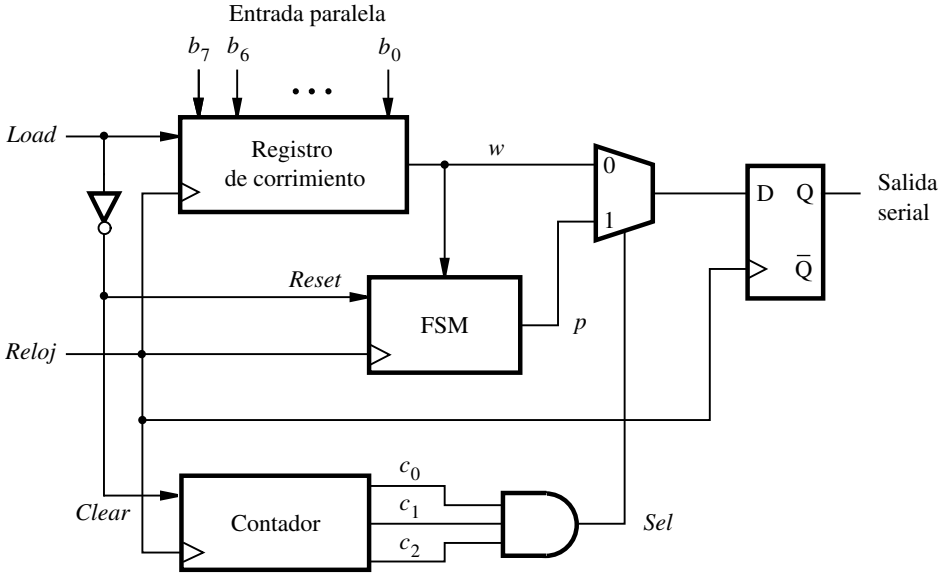


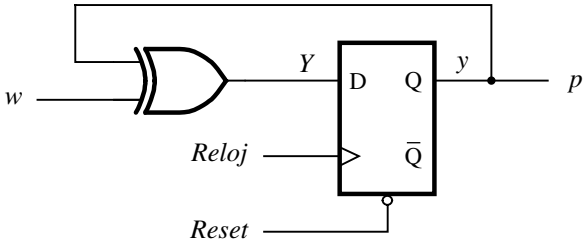
Figura 8.102 Convertidor paralelo a serial.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	p
S_{par}	S_{par}	S_{impar}	0
S_{impar}	S_{impar}	S_{par}	1

a) Tabla de estado

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	p
y	Y	Y	
0	0	1	0
1	1	0	1

b) Tabla de asignación de estados



c) Circuito

Figura 8.103 FSM para generación de paridad.

PROBLEMAS

Al final del libro se proporcionan las respuestas a los problemas marcados con asterisco.

***8.1** Una FSM se define mediante la tabla de asignación de estados de la figura P8.1. Derive un circuito que produzca esta FSM usando flip-flops D.

***8.2** Derive un circuito que realice la FSM definida por la tabla de asignación de estados de la figura P8.1 utilizando flip-flops JK.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	
$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
0 0	1 0	1 1	0
0 1	0 1	0 0	0
1 0	1 1	0 0	0
1 1	1 0	0 1	1

Figura P8.1 Tabla de asignación de estados para los problemas 8.1 y 8.2.

8.3 Derive un diagrama de estado para una FSM que tiene una entrada w y una salida z. La máquina debe generar $z = 1$ cuando los valores previos de w fueron 1001 o 1111; de otro modo, $z = 0$. Se permite sobreponer patrones de entrada. Un ejemplo del comportamiento buscado es

$$w : 010111100110011111$$
$$z : 000000100100010011$$

8.4 Escriba código de VHDL para la FSM descrita en el problema 8.3.

***8.5** Elabore una tabla de estados mínimos para una FSM tipo Moore de entrada y salida únicas que genera una salida de 1 si en la secuencia de entrada detecta patrones 110 o 101. Deben detectarse secuencias sobrepuestas.

***8.6** Repita el problema 8.5 para una FSM tipo Mealy.

8.7 Derive los circuitos que se aplican en las tablas de estado de las figuras 8.51 y 8.52. ¿Cuál es el efecto de la minimización de estados en el costo de la implementación?

8.8 Dibuje los circuitos que implementan las tablas de estado de las figuras 8.55 y 8.56. Compare los costos de los circuitos.

8.9 Un circuito secuencial tiene dos entradas, w_1 y w_2, y una salida, z. Su función es comparar las secuencias de entrada de las dos entradas. Si $w_1 = w_2$ en cualquiera de cuatro ciclos consecutivos de reloj, el circuito genera $z = 1$; de lo contrario $z = 0$. Por ejemplo

$$w_1 : 0110111000110$$
$$w_2 : 1110101000111$$
$$z : 0000100001110$$

Derive un circuito posible.

8.10 Escriba código de VHDL para la FSM descrita en el problema 8.9.

8.11 Una FSM tiene una entrada, w, y una salida, z. En cuatro pulsos consecutivos de reloj se aplica una secuencia de cuatro valores de la señal w. Calcule la tabla de estado para la FSM que genera $z = 1$ cuando detecta que se han aplicado las secuencias $w : 0010$ o $w : 1110$; de lo contrario, $z = 0$. Después del cuarto pulso de reloj, la máquina debe entrar de nuevo en el estado reset y estar lista para la secuencia siguiente. Minimice el número de estados necesarios.

***8.12** Construya una tabla de estados mínimos para una FSM que actúa como un generador de paridad de tres bits. Por cada tres bits que se observan en la entrada w durante tres ciclos consecutivos de reloj, la FSM genera el bit de paridad $p = 1$ si y sólo si el número de unos (1) en la secuencia de tres bits es impar.

8.13 Escriba código de VHDL para la FSM descrita en el problema 8.12.

8.14 Dibuje los diagramas de tiempo para los circuitos de las figuras 8.43 y 8.47, si se supone el mismo cambio en las señales a y b para ambos circuitos. Considere los retrasos de propagación.

***8.15** Muestre una tabla de estado para la tabla de asignación de estados de la figura P8.1. Emplee A, B, C y D para las cuatro filas de la tabla. Elabore una nueva tabla de asignación de estados con codificación de 1 activo. Para A utilice el código $y_4 y_3 y_2 y_1 = 0001$. Para los estados B, C y D emplee los códigos 0010, 0100 y 1000, respectivamente. Sintetice un circuito con flip-flops D.

8.16 Muestre cómo puede modificarse el circuito del problema 8.15, de modo que el código $y_4 y_3 y_2 y_1 = 0000$ se emplee para el estado reset, A, y los códigos para los estados B, C y D se cambien como sea necesario. (*Sugerencia*: no es necesario volver a sintetizar el circuito.)

***8.17** En la figura 8.59 suponga que las salidas no especificadas en los estados B y G son, respectivamente, 0 y 1. Construya la tabla de estados mínimos para esta FSM.

8.18 En la figura 8.59 suponga que las salidas no especificadas en los estados B y G son, respectivamente, 1 y 0. Construya la tabla de estados mínimos para esta FSM.

8.19 Dibuje los circuitos que implementan las FSM definidas en las figuras 8.57 y 8.58. ¿Puede concluir algo respecto a la complejidad de los circuitos que se aplican en los tipos de máquinas Moore y Mealy?

8.20 Diseñe un contador que cuente los pulsos de la línea w y despliegue el conteo en la secuencia 0, 2, 1, 3, 0, 2,... Utilice flip-flops D en el circuito.

***8.21** Repita el problema 8.20 con flip-flops JK.

***8.22** Repita el problema 8.20 con flip-flops T.

8.23 Diseñe un contador módulo 6 que cuente en la secuencia 0, 1, 2, 3, 4, 5, 0, 1,... El contador debe contar los pulsos del reloj si su entrada enable, w, es igual a 1. Utilice flip-flops D en el circuito.

8.24 Repita el problema 8.23 con flip-flops JK.

8.25 Repita el problema 8.23 con flip-flops T.

8.26 Diseñe un circuito de tres bits tipo contador controlado por la entrada w. Si $w = 1$, entonces el contador agrega 2 al contenido y lo envuelve si el conteo alcanza 8 o 9. Así, si el estado presente es 8 o 9, entonces el estado siguiente se convierte respectivamente en 0 o 1. Si $w = 0$, entonces el contador resta 1 del contenido y actúa como un contador descendente normal. Utilice flip-flops D en el circuito.

8.27 Repita el problema 8.26 con flip-flops JK.

8.28 Repita el problema 8.26 con flip-flops T.

***8.29** Derive la tabla de estado para el circuito de la figura P8.2. ¿Qué secuencia de valores de entrada en el cable *w* se detecta con este circuito?

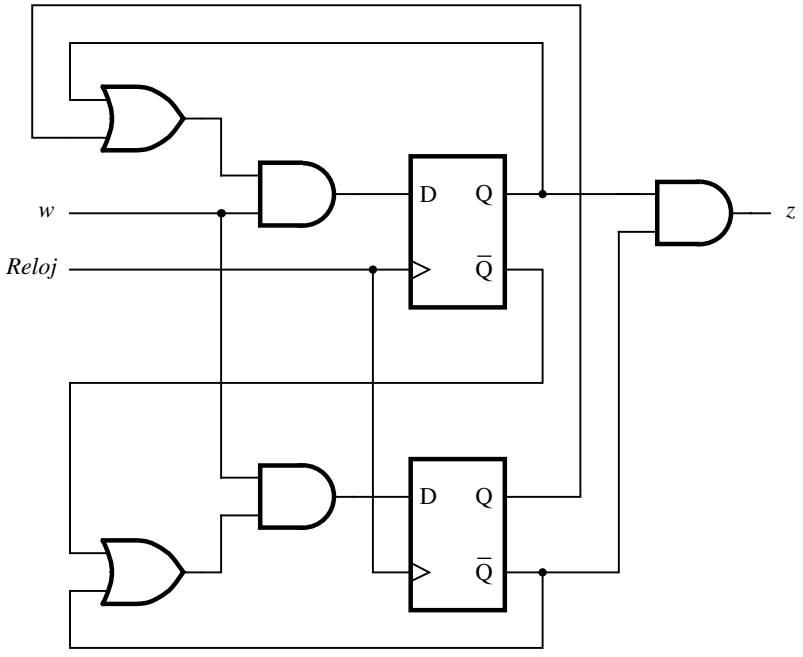


Figura P8.2 Circuito para el problema 8.29.

8.30 Escriba código de VHDL para la FSM mostrada en la figura 8.57, con el estilo del código de la figura 8.29.

8.31 Repita el problema 8.30, con el estilo del código de la figura 8.33.

8.32 Escriba código de VHDL para la FSM que se muestra en la figura 8.58, con el estilo del código de la figura 8.29.

8.33 Repita el problema 8.32, con el estilo del código de la figura 8.33.

8.34 Escriba código de VHDL para la FSM mostrada en la figura P8.1. Utilice el método de asignación de estados mostrado en la figura 8.34.

8.35 Repita el problema 8.34 con el método de asignación de estados mostrado en la figura 8.35.

8.36 Represente la FSM de la figura 8.57 en forma de carta ASM.

8.37 Represente la FSM de la figura 8.58 en forma de carta ASM.

8.38 La FSM árbitro definida en la sección 8.8 (figura 8.72) puede hacer que el dispositivo 3 nunca reciba servicio si los dispositivos 1 y 2 se mantienen continuamente haciendo solicitudes, de modo que en el estado *Idle* siempre sucede que el dispositivo 1 o 2 tiene una solicitud en puerta. Modifique la FSM pro-

puesta para asegurar que el dispositivo 3 tenga servicio, de tal forma que si se presenta una solicitud, los dispositivos 1 y 2 sólo recibirán servicio una vez antes que se conceda una solicitud al dispositivo 3.

8.39 Escriba código de VHDL para la FSM del problema 8.38.

8.40 Piense en una versión más general de la tarea presentada en el ejemplo 8.1. Suponga que hay cuatro registros de n bits conectados a un bus en un procesador. El contenido del registro R se coloca en el bus al validar la señal de control R_{out}. Los datos del bus se cargan en el flanco activo de la señal de reloj del registro R si la señal de control R_{in} se valida. Suponga que se emplean como registros normales tres de los registros, llamados $R1$, $R2$ y $R3$. El cuarto registro, *TEMP*, se emplea para almacenamiento temporal en casos especiales.
Se quiere realizar la operación SWAP Ri,Rj, que intercambia el contenido de los registros *Ri* y *Rj*. Esto se logra mediante la secuencia de pasos siguiente (cada uno realizado en un ciclo de reloj)

$$\begin{aligned} \text{TEMP} &\leftarrow [\text{Rj}] \\ \text{Rj} &\leftarrow [\text{Ri}] \\ \text{Ri} &\leftarrow [\text{TEMP}] \end{aligned}$$

Dos señales de entrada, w_1 y w_2, se usan para indicar que dos registros deben intercambiarse de la manera siguiente

Si $w_2w_1 = 01$, entonces intercámbiense $R1$ y $R2$.
Si $w_2w_1 = 10$, entonces intercámbiense $R1$ y $R3$.
Si $w_2w_1 = 11$, entonces intercámbiense $R2$ y $R3$.

Una combinación de entrada que especifica un intercambio debe durar tres ciclos de reloj. Diseñe un circuito que genere las señales de control requeridas: $R1_{out}$, $R1_{in}$, $R2_{out}$, $R2_{in}$, $R3_{out}$, $R3_{in}$, $TEMP_{out}$ y $TEMP_{in}$. Derive las expresiones de estado siguiente y de salida para este circuito, intentando minimizar el costo.

8.41 Escriba código de VHDL para describir el circuito de la figura 8.102.

8.42 En la sección 8.5 se presenta un diseño para el sumador serial. Dibuje un circuito parecido que funcione como un restador serial que genere la diferencia de los operandos A y B. (*Sugerencia*: utilice la regla para encontrar complementos a 2, sección 5.3.1, para generar el complemento a 2 de B.)

8.43 Escriba código de VHDL que defina el restador serial diseñado en el problema 8.42.

BIBLIOGRAFÍA

1. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: 1997).
2. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

3. M. M. Mano, *Digital Design*, 3a. ed. (Prentice-Hall: Upper Saddle River, NJ, 2002).
4. J. P. Daniels, *Digital Design from Zero to One* (Wiley: Nueva York, 1996).
5. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
6. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
7. F. J. Hill y G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4a. ed. (Wiley: Nueva York, 1993).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4a. ed. (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
11. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. T. L. Booth, *Digital Networks and Computer Systems* (Wiley: Nueva York, 1971).
13. Z. Kohavi, *Switching and Finite Automata Theory* (McGraw-Hill: Nueva York, 1970).
14. J. Hartmanis y R. E. Stearns, *Algebraic Structure Theory of Sequential Machines* (Prentice-Hall: Englewood Cliffs, NJ, 1966).

capítulo

9

CIRCUITOS SECUENCIALES ASÍNCRONOS

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

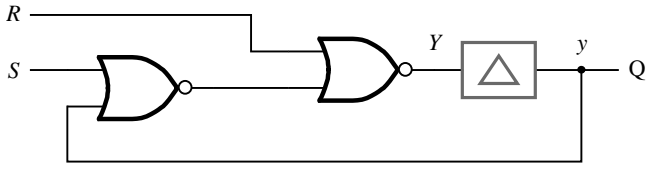
- Circuitos secuenciales no sincronizados por un reloj
- Análisis de los circuitos secuenciales asíncronos
- Síntesis de los circuitos secuenciales asíncronos
- El concepto de estado estable e inestable
- Riesgos que provocan el comportamiento incorrecto de un circuito

En el capítulo anterior expusimos el diseño de los circuitos secuenciales síncronos, en los que las variables de estado se representan por medio de flip-flops controlados por un reloj. El reloj es una señal periódica que consiste en pulsos. Los cambios en el estado pueden ocurrir en el flanco positivo o negativo de cada pulso de reloj. Como están controlados por medio de pulsos, se dice que los circuitos secuenciales síncronos operan en *modo de pulso*. En este capítulo estudiamos circuitos secuenciales que no operan en tal modo ni utilizan flip-flops para representar las variables de estado. Estos circuitos se llaman *circuitos secuenciales asíncronos*.

En un circuito secuencial asíncrono, los cambios en el estado no se disparan por pulsos de reloj sino que dependen de si las entradas al circuito tiene el nivel lógico 0 o 1 en un momento específico. Para lograr la operación confiable, las entradas al circuito deben cambiar una por una. Además, debe haber suficiente tiempo entre los cambios en las señales de entrada para que el circuito llegue a un *estado estable*, el cual se logra cuando todas las señales internas dejan de cambiar. Se dice que un circuito que observa estas restricciones opera en el *modo fundamental*.

9.1 COMPORTAMIENTO ASÍNCRONO

Para estudiar los circuitos secuenciales asíncronos se reconsiderará el circuito de latch básico de la figura 7.4. Ese latch Set-Reset (SR) se volvió a trazar en la figura 9.1*a*. El lazo de retroalimentación da lugar a su naturaleza secuencial. Se trata de un circuito asíncrono porque los cambios



a) Circuito con retraso de compuerta modelado

Estado presente y	Estado siguiente			
	$SR = 00$	01	10	11
	Y	Y	Y	Y
0	(0)	(0)	1	(0)
1	(1)	0	(1)	0

b) Tabla de asignación de estados

Figura 9.1 Análisis del latch SR.

en el valor de la salida, Q, ocurren sin necesidad de esperar un pulso de reloj que lleve a cabo la sincronización. En respuesta a un cambio ya sea en la entrada *S* (Set) o en *R* (Reset), el valor de Q cambiará después de un breve tiempo de propagación a través de las compuertas NOR. En la figura 9.1*a* el retraso de propagación combinado por las dos compuertas NOR se representa mediante un cuadro etiquetado Δ. Por tanto, los símbolos de la compuerta NOR representan compuertas ideales con un retraso de cero. Si empleamos la notación del capítulo 8, Q corresponde al *estado presente* del circuito, representado por la *variable del estado presente*, *y*. El valor de *y* se alimenta de nuevo a través del circuito para generar el valor de la *variable del estado siguiente*, *Y*, que representa el *estado siguiente* del circuito. Después del retardo de tiempo Δ, *y* toma el valor de *Y*. Obsérvese que hemos dibujado el circuito en un estilo que se ajusta al modelo general de los circuitos secuenciales presentado en la figura 8.90.

Al analizar el latch SR, podemos derivar una tabla de asignación de estados, como se ilustra en la figura 9.1*b*. Cuando el estado presente es $y = 0$ y las entradas son $S = R = 0$, el circuito produce $Y = 0$. Como $y = Y$, el estado del circuito no cambiará. Decimos que el circuito es *estable* en estas condiciones de entrada. Ahora supóngase que *R* cambia a 1 mientras *S* permanece en 0. El circuito aún genera $Y = 0$ y sigue estable. Digamos ahora que *S* cambia a 1 y *R* permanece en 1. El valor de *Y* sigue sin cambiar y el circuito es estable. Entonces *R* cambia a 0 mientras *S* permanece en 1. Esta combinación de entrada, $SR = 10$, hace que el circuito genere $Y = 1$. Como $y \neq Y$, el circuito no es estable. Después del retraso de tiempo Δ, el circuito cambia al nuevo estado presente $y = 1$. Una vez que se alcanza este estado nuevo, el valor de *Y* sigue siendo igual a 1 mientras $SR = 10$. Por consiguiente, el circuito de nuevo es estable. El análisis para el estado presente $y = 1$ puede completarse siguiendo un razonamiento similar.

El concepto de estados estables es muy importante en el contexto de los circuitos secuenciales asíncronos. Para una combinación de las entradas, si un circuito alcanza un estado en particular y permanece en este estado, entonces se dice que el estado es estable. Para indicar claramente las condiciones en las que un circuito es estable se acostumbra encerrar en un círculo los estados estables en la tabla, como se muestra en la figura 9.1*b*.

A partir de la tabla de asignación de estados podemos derivar la tabla de estado de la figura 9.2*a*. Los nombres de estado *A* y *B* representan los estados presentes $y = 0$ y $y = 1$, respectivamente. Puesto que la salida Q sólo depende del valor presente, el circuito es una FSM tipo Moore. El diagrama de estado que representa el comportamiento de esta FSM se muestra en la figura 9.2*b*.

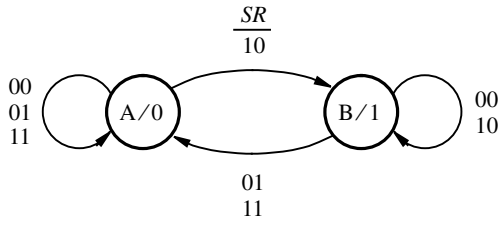
El análisis anterior muestra que el comportamiento de un circuito secuencial asíncrono puede representarse como una FSM de una manera muy semejante a los circuitos secuenciales síncronos del capítulo 8. Considérese ahora realizar la tarea contraria. Es decir, dada la tabla de estado de la figura 9.2*a*, podemos sintetizar un circuito asíncrono como sigue: tras realizar la asignación de estados tenemos la tabla de asignación de estados de la figura 9.1*b*. Esta tabla representa una tabla de verdad para *Y*, con las entradas *y*, *S* y *R*. La derivación de una expresión mínima de producto de las sumas genera

$$Y = \overline{R} \cdot (S + y)$$

Si estuviéramos derivando un circuito secuencial síncrono aplicando los métodos del capítulo 8, entonces estaríamos conectados a la entrada *D* de un flip-flop y se utilizaría una señal de reloj para controlar el tiempo cuando los cambios de estado ocurran. Pero como estamos sintetizando un circuito asíncrono, no insertamos un flip-flop en la trayectoria de retroalimentación. En vez de ello, creamos un circuito que produce la expresión anterior usando las compuertas lógicas

Estado presente	Estado siguiente				Salida Q
	$SR = 00$	01	10	11	
A	(A)	(A)	B	(A)	0
B	(B)	A	(B)	A	1

a) Tabla de estado



b) Diagrama de estado

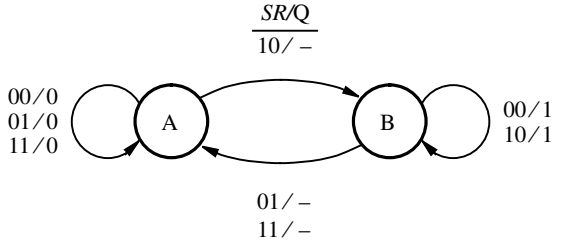
Figura 9.2 Modelo de una FSM para el latch SR.

necesarias, y alimentamos de nuevo la señal de salida como la entrada del estado presente y. La implementación con compuertas NOR da como resultado el circuito de la figura 9.1*a*. Este ejemplo simple sugiere que los circuitos asíncronos y los circuitos síncronos pueden sintetizarse utilizando técnicas parecidas. Sin embargo, veremos en breve que la tarea de diseño es considerablemente más difícil para circuitos asíncronos más complejos.

Para explorar más la naturaleza de los circuitos asíncronos, es interesante considerar cómo el comportamiento del latch SR puede representarse con un modelo Mealy. Como se describe en la figura 9.3, las salidas producidas cuando el circuito está en un estado estable son las mismas que en el modelo Moore, a saber, 0 en el estado A y 1 en el B. Considérese ahora lo que ocurre cuando el estado del circuito cambia. Supóngase que el estado presente es A y que la combinación de entrada SR cambia de 00 a 10. Como la tabla de estado lo especifica, el estado siguiente de la FSM es B. Cuando el circuito alcanza el estado B, la salida Q será 1. Pero en el modelo Mealy se supone que la salida se verá afectada de inmediato por un cambio en las señales de entrada. Por tanto, mientras sigue en el estado A, el cambio en SR a 10 debe resultar en Q = 1. Podríamos haber escrito un 1 en la entrada correspondiente en la fila superior de la tabla de estado, pero hemos elegido dejarla sin especificar porque como Q cambiará a 1 en cuanto el circuito entre en el estado B, no se gana mucho si se logra que Q pase a 1 un poco antes. Dejar la entrada sin especificar nos permite asignarle 0 o 1, lo cual tal vez simplifique un poco el circuito que implementa la tabla de estado. Un razonamiento similar nos lleva a la conclusión

Estado presente	Estado siguiente				Salida, Q			
	$SR = 00$	01	10	11	00	01	10	11
A	(A)	(A)	B	(A)	0	0	–	0
B	(B)	A	(B)	A	1	–	1	–

a) Tabla de estado



b) Diagrama de estado

Figura 9.3 Representación Mealy del latch SR.

de que los dos lugares de salida donde ocurre un cambio de B a A también pueden dejarse sin especificar.

Usando la asignación de estados $y = 0$ para A y $y = 1$ para B, la tabla de asignación de estados representa una tabla de verdad tanto para Y como para Q. La expresión mínima para Y es la misma que para el modelo Moore. Para derivar una expresión para Q debemos establecer en 0 o 1 las entradas sin especificar. Si asignamos un 0 a la entrada sin especificar en la primera fila y un 1 a las entradas sin especificar en la segunda resultará en $Q = y$ y eso producirá el circuito de la figura 9.1*a*.

Terminología

En la exposición precedente usamos la misma terminología que en el capítulo anterior, que versó sobre los circuitos secuenciales síncronos. No obstante, cuando se trata de circuitos secuenciales asíncronos es costumbre utilizar dos términos distintos. En vez de una "tabla de estado" es más común hablar de una *tabla de flujo*, la cual explica los cambios en el flujo de estado como resultado de los cambios en las señales de entrada. En vez de una "tabla de asignación de estados", es costumbre referirse a una *tabla de transición* o a una *tabla de excitación*. En este capítulo usaremos los términos *tabla de flujo* y *tabla de excitación*. Una tabla de flujo definirá los cambios de estado y las salidas que deben generarse. Una tabla de excitación representará las transiciones en términos de las variables de estado. El término *tabla de excitación* proviene del hecho de que un cambio de un estado estable se realiza al "excitar" las variables del estado siguiente para comenzar a cambiar al estado nuevo.

9.2 ANÁLISIS DE LOS CIRCUITOS ASÍNCRONOS

Para familiarizarse con los circuitos asíncronos es útil analizar algunos ejemplos. Tendremos en mente el modelo general de la figura 8.90, suponiendo que los retardos en las rutas de retroalimentación son una representación de los retrasos de propagación en el circuito. Por tanto, cada símbolo de compuerta representará una compuerta ideal con cero retraso.

Ejemplo 9.1 **LATCH D ASÍNCRONO** En los capítulos 7 y 8 utilizamos el latch D asíncrono como componente central en los circuitos controlados por un reloj de sincronización. Es conveniente analizar este latch como un circuito asíncrono, donde el reloj es sólo una de las entradas. Es razonable suponer que las señales en las entradas D y de reloj no cambian al mismo tiempo, con lo que se cumplen los requisitos básicos de los circuitos asíncronos.

En la figura 9.4*a* se muestra el latch D asíncrono dibujado en el estilo del modelo de la figura 8.90. Este circuito se presentó en la figura 7.8 y se estudió en la sección 7.3. La expresión de estado siguiente para este circuito es

$$\begin{aligned} Y &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y) \\ &= CD + \overline{C}y + Dy \end{aligned}$$

El término Dy en esta expresión es redundante y podría eliminarse sin cambiar la función lógica de Y. Por consiguiente la expresión mínima es

$$Y = CD + \overline{C}y$$

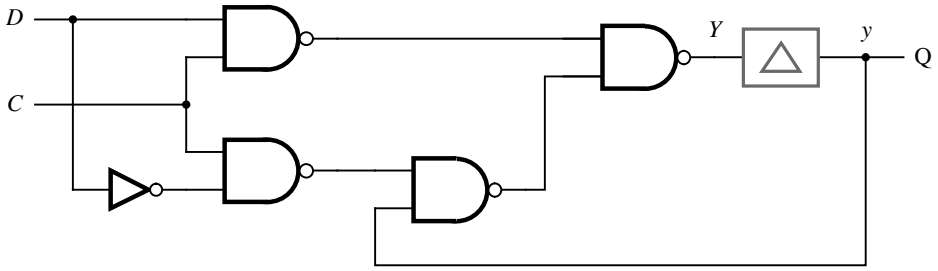
La razón por la que el circuito implementa el término redundante Dy es que este término resuelve una "condición de carrera" conocida como *riesgo*; estudiaremos los riesgos con detalle en la sección 9.6.

La evaluación de la expresión para Y para todas las combinaciones de C, D y y conduce a la tabla de excitación de la figura 9.4*b*. Observe que el circuito cambia su estado sólo cuando $C = 1$ y D es diferente del estado presente, y. En todos los demás casos el circuito es estable. Usando los símbolos A y B para representar los estados $y = 0$ y $y = 1$ obtenemos la tabla de flujo y el diagrama de estado mostrados en los incisos (*c*) y (*d*).

Ejemplo 9.2 **FLIP-FLOP D MAESTRO-ESCLAVO** En el ejemplo 9.1 analizamos el latch D asíncrono. En realidad, todos los circuitos prácticos son asíncronos. Sin embargo, si el comportamiento del circuito está rigurosamente controlado por una señal de reloj, entonces pueden usarse suposiciones de operación más simples, como lo hicimos en el capítulo 8. Recuérdese que en un circuito secuencial síncrono todas las señales cambian de valor en sincronización con la señal del reloj. Ahora examinaremos otro circuito síncrono como si fuera un circuito asíncrono.

Dos latches D asíncronos se usan para implementar el flip-flop D maestro-esclavo, como se ilustra en la figura 7.10. Este circuito se reproduce en la figura 9.5. Podemos analizar el circuito tratándolo como una conexión en serie de dos latches D asíncronos. Con los resultados del ejemplo 9.1, las expresiones simplificadas de estado siguiente pueden escribirse como

$$\begin{aligned} Y_m &= CD + \overline{C}y_m \\ Y_s &= \overline{C}y_m + Cy_s \end{aligned}$$



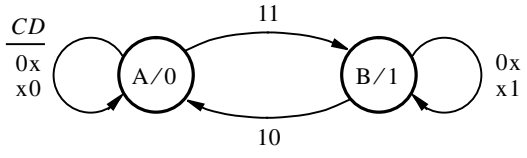
a) Circuito

Estado presente y	Estado siguiente $CD = 00$	01	10	11	Q
	Y	Y	Y	Y	
0	(0)	(0)	(0)	1	0
1	(1)	(1)	0	(1)	1

b) Tabla de excitación

Estado presente	Estado siguiente $CD = 00$	01	10	11	Q
A	(A)	(A)	(A)	B	0
B	(B)	(B)	A	(B)	1

c) Tabla de flujo



d) Diagrama de estado

Figura 9.4 El latch D asíncrono.

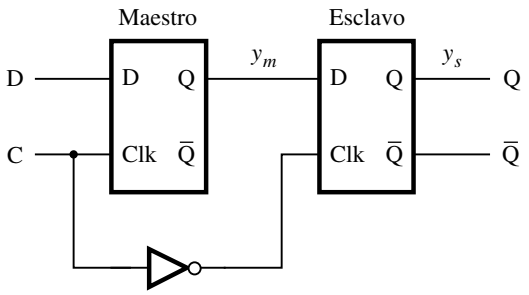


Figura 9.5 Circuito para el flip-flop D maestro-esclavo.

donde los subíndices *m* y *s* se refieren a las etapas de maestro y esclavo del flip-flop. Estas expresiones conducen a la tabla de excitación de la figura 9.6*a*. Al etiquetar los cuatro estados de *S*1 a *S*4 derivamos la tabla de flujo de la figura 9.6*b*. Un diagrama de estado que representa esta información se da en la figura 9.7.

Consideremos el comportamiento de esta FSM con más detalle. El estado *S*1, donde $y_m y_s = 00$, es estable para todas las combinaciones de entrada excepto $CD = 11$. Cuando $C = 1$, el valor de *D* se almacena en la etapa de maestro; por consiguiente, $CD = 11$ hace que el flip-flop cambie a *S*3, donde $y_m = 1$ y $y_s = 0$. Si la entrada *D* ahora cambia de nuevo a 0, mientras el reloj permanece en 1, el flip-flop vuelve al estado *S*1. Las transiciones entre *S*1 y *S*3 indican que si $C = 1$, la salida de la etapa de maestro, $Q_m = y_m$, rastrea los cambios en la señal de entrada *D* sin afectar la etapa de esclavo. De *S*3 el circuito cambia a *S*4 cuando el reloj pasa a 0. En *S*4 las dos etapas, maestro y esclavo, se establecen en 1 porque la información de la etapa de maestro se transfiere a la de esclavo en el flanco negativo del reloj. Ahora el flip-flop se queda en *S*4 hasta que el reloj pasa a 1 y la entrada *D* cambia a 0, lo que produce un cambio a *S*2. En *S*2 la etapa de maestro se borra a 0, pero la de esclavo permanece en 1. De nuevo el flip-flop puede cambiar entre *S*2 y *S*4 debido a que la etapa de maestro rastreará los cambios en la señal de entrada *D* mientras $C = 1$. De *S*2 el circuito cambia a *S*1 cuando el reloj pasa a nivel bajo.

En las figuras 9.6 y 9.7 indicamos que el flip-flop tiene sólo una salida Q, que se observa cuando el circuito se ve como un flip-flop disparado por flanco negativo. Desde el punto de vista del observador, el flip-flop tiene sólo dos estados, 0 y 1, pero internamente consta de las partes maestro y esclavo, lo que da origen a los cuatro estados descritos líneas arriba.

También debemos examinar la suposición básica de que las entradas deben cambiar una por una. Si el circuito es estable en el estado *S*2, donde $CD = 10$, es imposible pasar de este estado a *S*1 por la influencia de la combinación de entrada $CD = 01$, ya que este cambio simultáneo en ambas entradas no puede ocurrir. Por tanto, en la segunda fila de la tabla de flujo, en vez de mostrar *S*2 que cambia a *S*1 cuando $CD = 01$, esta entrada puede etiquetarse como sin especificar. El cambio de *S*2 a *S*1 puede ser ocasionado sólo por el cambio de *CD* de 10 a 00. De manera similar, si el circuito está en el estado *S*3, donde $CD = 11$, no puede cambiar a *S*4 al tener $CD = 00$. Esta entrada también puede dejarse sin especificar en la tabla. La tabla de flujo resultante se muestra en la figura 9.6*c*.

Si invertimos el procedimiento de análisis y, mediante la asignación de estados de la figura 9.6*a*, sintetizamos las expresiones lógicas para Y_m y Y_s, obtenemos

$$Y_m = CD + \overline{C}y_m + y_m D$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

Estado presente $y_m y_s$	Estado siguiente $CD = 00$	01	10	11	Salida Q
	$Y_m Y_s$				
00	(00)	(00)	(00)	10	0
01	00	00	(01)	11	1
10	11	11	00	(10)	0
11	(11)	(11)	01	(11)	1

a) Tabla de excitación

Estado presente	Estado siguiente $CD = 00$	01	10	11	Salida Q
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	S1	(S2)	S4	1
S3	S4	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

b) Tabla de flujo

Estado presente	Estado siguiente $CD = 00$	01	10	11	Salida Q
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	—	(S2)	S4	1
S3	—	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

c) Tabla de flujo con entradas sin especificar

Figura 9.6 Tablas de excitación y de flujo para el ejemplo 9.2.

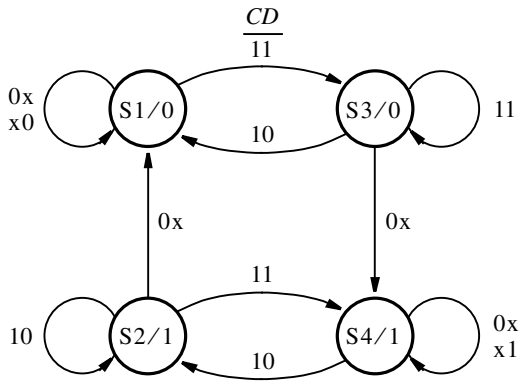


Figura 9.7 Diagrama de estado para el flip-flop D maestro-esclavo.

Los términos $y_m D$ y $y_m y_s$ en estas expresiones son redundantes. Como mencionamos antes, se incluyen en el circuito para evitar condiciones de carrera, las cuales se estudian en la sección 9.6.

Ejemplo 9.3 Considere el circuito de la figura 9.8. Está representado por las expresiones siguientes

$$Y_1 = y_1\bar{y}_2 + w_1\bar{y}_2 + \overline{w}_1\overline{w}_2 y_1$$
$$Y_2 = y_1 y_2 + w_1 y_2 + w_2 + \overline{w}_1\overline{w}_2 y_1$$
$$z = \bar{y}_1 y_2$$

Las tablas de excitación y de flujo correspondientes se presentan en la figura 9.9.

Algunas transiciones en la tabla de flujo no ocurrirán en la práctica debido a la suposición de que tanto w_1 como w_2 no pueden cambiar al mismo tiempo. En el estado A el circuito es estable bajo la combinación $w_2w_1 = 00$. Sus entradas no pueden cambiar a 11 sin pasar por las combinaciones 01 o 10, caso en el que el estado nuevo sería B o C, respectivamente. Por tanto, la transición de A bajo $w_2w_1 = 11$ puede dejarse sin especificar. De forma similar, si el circuito es estable en el estado B, en cuyo caso $w_2w_1 = 01$, es imposible forzar un cambio al estado D al cambiar las entradas a $w_2w_1 = 10$. Este lugar también debe quedarse sin especificar. Si el circuito es estable en el estado C bajo $w_2w_1 = 11$, no es posible pasar a A cambiando las entradas directamente a $w_2w_1 = 00$. Sin embargo, la transición a A es posible si se cambian las entradas una por una, ya que el circuito permanece estable en C tanto para $w_2w_1 = 01$ como para $w_2w_1 = 10$.

Una situación diferente surge si el circuito es estable en el estado D bajo $w_2w_1 = 00$. Tal vez parezca que la transición bajo $w_2w_1 = 11$ debe estar sin especificar porque no puede hacerse un cambio en esta entrada desde el estado estable D. Pero suponga que el circuito es estable en el estado B bajo $w_2w_1 = 01$. Ahora supongamos que las entradas cambian a $w_2w_1 = 11$. Esto ocasiona un cambio hacia el estado D. El circuito de hecho cambia a D, pero no es estable en este estado para su condición de entrada. En cuanto llega al estado D, el circuito procede a cambiar al estado C según lo requiere $w_2w_1 = 11$. Entonces está estable en el estado C siempre que ambas entradas permanezcan en 1. La conclusión es que la transición que especifica el cambio de D a C

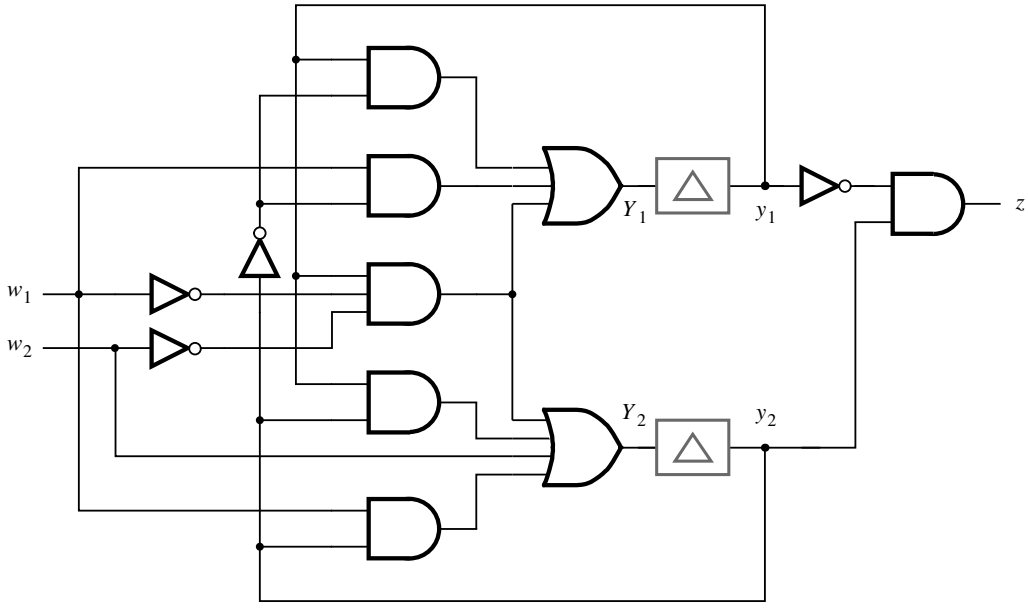


Figura 9.8 Circuito para el ejemplo 9.3.

bajo $w_2w_1 = 11$ es significativa y no debe omitirse. La transición del estado estable *B* al estado estable *C*, la cual pasa por el estado *D*, ilustra que no es imperativo que todas las transiciones ocurran directamente de un estado estable a otro. Un estado por el que un circuito pasa en su ruta de un estado estable a otro se llama *estado inestable*. Las transiciones que suponen el paso por un estado instable no son tan dañinas siempre que el estado inestable no genere una señal de salida indeseable. Por ejemplo, si una transición está entre dos estados estables para los que la señal de salida debe ser 0, sería inaceptable pasar por un estado inestable que ocasione que la salida sea 1. Aun cuando el circuito cambia a través del estado inestable muy rápido, es probable que la breve oscilación en la señal de salida provoque dificultades. Esto no es así en nuestro ejemplo. Cuando el circuito está estable en *B*, la salida es $z = 0$. Cuando las entradas cambian a $w_2w_1 = 11$, la transición al estado *D* mantiene la salida en 0. Es sólo cuando el circuito finalmente cambia al estado *C* que z cambiará a 1. Por tanto, el cambio de $z = 0$ a $z = 1$ ocurre únicamente una vez durante el curso de estas transiciones.

En la figura 9.10 se presenta una tabla de flujo modificada, que muestra las transiciones sin especificar. La tabla indica el comportamiento del circuito de la figura 9.8 en términos de las transiciones de estado. Si no sabemos qué se supone que hace el circuito, será difícil descubrir su aplicación práctica. Por fortuna, en la vida real el propósito del circuito es conocido y el diseñador realiza el análisis para determinar que su desempeño sea el buscado. En nuestro ejemplo es evidente que el circuito genera la salida $z = 1$ en el estado *C*, que alcanza como resultado de algunos patrones de entrada que se detectan usando los otros tres estados. El diagrama de estado derivado de la figura 9.10 se muestra en la figura 9.11.

Este diagrama en realidad implementa un mecanismo de control para una máquina expendedora simple que acepta dos tipos de monedas, por ejemplo de 5 y 10 centavos, y despacha

Estado presente y_2y_1	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	(00)	01	10	11	0
01	11	(01)	11	11	0
10	00	(10)	(10)	(10)	1
11	(11)	10	10	10	0

a) Tabla de excitación

Estado presente	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
A	(A)	B	C	D	0
B	D	(B)	D	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

b) Tabla de flujo

Figura 9.9 Tablas de excitación y de flujo para el circuito de la figura 9.8.

Estado presente	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
A	(A)	B	C	–	0
B	D	(B)	–	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

Figura 9.10 Tabla de flujo modificada para el ejemplo 9.3.

mercancía como caramelos. Si w_1 representa una moneda de cinco centavos y w_2 una de 10, entonces ha de depositarse un total de 10 centavos para llevar a la FSM al estado C, donde se suelta el caramelo. El mecanismo tragamonedas acepta sólo una moneda a la vez, lo que significa que $w_2w_1 = 11$ nunca puede ocurrir. Por consiguiente, la transición estudiada líneas arriba, de B a C, a través del estado instable D no ocurriría. Observe que ambos estados B y D señalan que se

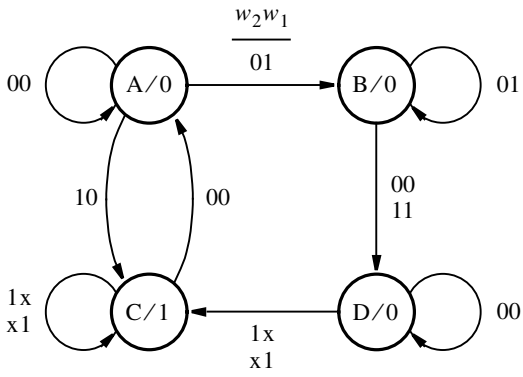


Figura 9.11 Diagrama de estado para el ejemplo 9.3.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	—	0
B	D	(B)	—	—	0
C	A	(C)	(C)	—	1
D	(D)	C	C	—	0

$w_2 \equiv 10$ centavos $w_1 \equiv 5$ centavos

Figura 9.12 Tabla de flujo para una máquina expendedora simple.

han depositado cinco centavos. El estado *B* indica que en ese momento el detector de monedas detecta una moneda de cinco centavos, en tanto que *D* dice que se han depositado cinco centavos y el receptor de monedas está vacío. En el estado *D* es posible depositar una moneda de 5 o una de 10 centavos; ambas acciones llevan al estado *C*. No se hace ninguna distinción entre los dos tipos de monedas en el estado *D*; por consiguiente, la máquina no daría cambio si se depositaran 15 centavos. Del estado *A* una moneda de 10 centavos conduce directamente al estado *C*. Saber que la condición $w_2w_1 = 11$ no ocurrirá permite que la tabla de flujo se especifique como se muestra en la figura 9.12. Si fuéramos a sintetizar las expresiones lógicas de la suma de productos para Y_1 y Y_2 usando la asignación de estados de la figura 9.9*a* terminaríamos con el circuito de la figura 9.8.

Pasos en el proceso de análisis

Hemos demostrado el proceso de análisis mediante ejemplos ilustrativos. Los pasos requeridos pueden resumirse como sigue:

- Un circuito se interpreta en la forma del modelo general de la figura 8.90. Es decir, cada trayectoria de retroalimentación se corta y se inserta un elemento de retraso en el punto

donde se hace el corte. La señal de entrada al elemento de retraso representa una variable de estado siguiente correspondiente, Y_i, mientras que la señal de salida es la variable de estado presente, y_i. Es posible hacer un corte en cualquier parte dentro de un lazo específico formado por la conexión de retroalimentación, siempre que sólo haya un corte por lazo (variable de estado). Por tanto, el número de cortes que deben realizarse es el número más pequeño que da como resultado la falta de retroalimentación en cualquier parte del circuito excepto desde la salida de un elemento de retraso. Este número mínimo de cortes a veces se conoce como *conjunto de cortes*. Nótese que el análisis basado en un corte hecho en un punto de un lazo tal vez no produzca la misma tabla de flujo que un análisis hecho sobre un corte realizado en otro punto de ese mismo lazo. Pero las dos tablas de flujo reflejarían el mismo comportamiento funcional en términos de las entradas aplicadas y las salidas generadas.

- Las expresiones de estado siguiente y de salida se derivan del circuito.
- Se construye la tabla de excitación que corresponde a las expresiones de estado siguiente y de salida.
- Se obtiene una tabla de flujo asociando algunos nombres (arbitrarios) con los estados codificados particulares.
- Se deriva un diagrama de estado correspondiente de la tabla de flujo, si se desea.

9.3 SÍNTESIS DE LOS CIRCUITOS ASÍNCRONOS

La síntesis de circuitos secuenciales asíncronos sigue los mismos pasos básicos usados para sintetizar circuitos síncronos, los cuales se estudian en el capítulo 8. Hay ciertas diferencias debidas a la naturaleza asíncrona, que vuelven los circuitos asíncronos más difíciles de diseñar. Explicaremos las diferencias mediante algunos ejemplos de diseño. Los pasos básicos son:

- Idear un diagrama de estado para una FSM que realice el comportamiento funcional requerido.
- Derivar la tabla de flujo y reducir el número de estados de ser posible.
- Realizar la asignación de estados y construir la tabla de excitación.
- Obtener las expresiones de estado siguiente y de salida.
- Construir un circuito que implemente estas expresiones.

Cuando se crea un diagrama de estado, o tal vez la tabla de flujo directamente, es esencial cerciorarse que cuando el circuito se halle en un estado estable se generen las señales de salida correctas. Si es necesario pasar por un estado inestable, éste no debe producir una señal de salida indeseable.

La minimización de estados no es sencilla. Un procedimiento para llevarla a cabo se describe en la sección 9.4.

La asignación de estados no se realiza con el único propósito de reducir el costo del circuito final. En los circuitos asíncronos ciertas asignaciones de estado pueden ocasionar que el circuito sea poco confiable. Explicaremos este problema usando los ejemplos siguientes.

Ejemplo 9.4

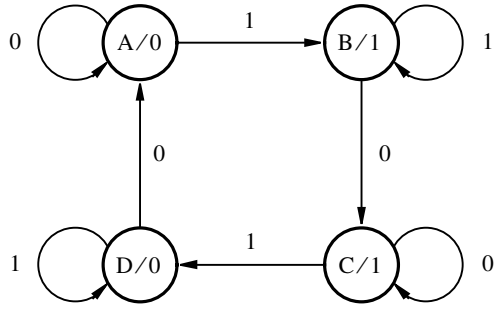
GENERADOR DE PARIDAD SERIAL Suponga que queremos diseñar un circuito que tiene una entrada w y una salida z tales que cuando se aplican pulsos a w, la salida z es igual a 0 si el número de pulsos aplicados previamente es par e igual a 1 si es non. Por consiguiente, el circuito actúa como un generador de paridad serial.

Sea A el estado que indica que se ha recibido un número par de pulsos. Usando el modelo Moore, la salida z será igual a 0 cuando el circuito esté en el estado A. Siempre que $w = 0$, el circuito debe permanecer en A, lo cual se especifica mediante un arco de transición que se origina y termina en ese estado. Por tanto, A es estable cuando $w = 0$. Cuando llega el pulso siguiente, la entrada $w = 1$ debe hacer que la FSM se mueva a un nuevo estado, digamos B, que produce la salida $z = 1$. Cuando la FSM llega a B debe permanecer estable en ese estado mientras $w = 1$. Esto se especifica por medio de un arco de transición que se origina y termina en B. El siguiente cambio en la entrada ocurre cuando w pasa a 0. En respuesta la FSM debe cambiar a un estado donde $z = 1$ y que corresponde al hecho de que un pulso completo se ha observado, es decir, que w ha cambiado de 1 a 0. Sea este estado C; debe ser estable bajo la condición de entrada $w = 0$. La llegada del pulso siguiente hace que $w = 1$, y la FSM debe cambiar al estado D, que indica que un número non de pulsos se ha observado y que el último pulso aún está presente. El estado D se halla estable bajo $w = 1$, y hace que la salida sea $z = 0$. Finalmente, cuando w regresa a 0 al final del pulso, la FSM vuelve al estado A, lo que indica un número par de pulsos y que w es igual a 0 en ese momento. El diagrama de estado resultante se muestra en la figura 9.13*a*.

Un punto clave que debe entenderse es por qué es necesario tener cuatro estados en vez de dos, considerando que simplemente estamos tratando de distinguir entre el número par e impar de pulsos de entrada. Los estados B y C no pueden combinarse en un solo estado aun cuando ambos indican que se ha observado un número impar de pulsos. Suponga que intentamos usar el estado B sólo para este propósito. Entonces habría sido necesario añadir un arco con la etiqueta 0 que empiece y termine en el estado B, lo cual está bien. El problema es que sin estado C tendría que haber una transición del estado B directamente a D si la entrada es $w = 1$ para responder al cambio siguiente en la entrada cuando llegue un pulso nuevo. Sería imposible hacer que B fuera estable bajo $w = 1$ y cambiara a D con la misma condición de entrada. De forma similar, podemos mostrar que los estados A y D no pueden combinarse en uno solo.

En la figura 9.13*b* se presenta la tabla de flujo que corresponde directamente al diagrama de estado. En muchos casos el diseñador puede derivar una tabla de flujo de manera directa. Estamos utilizando el diagrama de estado principalmente porque ofrece una imagen visual más simple del efecto de las transiciones de una FSM.

El paso siguiente consiste en asignar los valores a los estados en términos de las variables de estado. Como existen cuatro estados en nuestra FSM, debe haber cuando menos dos variables de estado. Sean estas variables y_1 y y_2. Como primer intento en la asignación de estados, sean los estados A, B, C y D codificados como $y_2y_1 = 00$, 01, 10 y 11, respectivamente. Esta asignación conduce a la tabla de excitación de la figura 9.14*a*. Por desgracia, tiene una falla importante. El circuito que implementa esta tabla es estable en el estado $D = 11$ bajo la condición de entrada $w = 1$. Pero considere lo que ocurre después si la entrada cambia a $w = 0$. De acuerdo con la tabla de excitación, el circuito debe cambiar al estado $A = 00$ y permanecer estable en él. El problema es que en el paso de $y_2y_1 = 11$ a $y_2y_1 = 00$ las dos variables de estado deben cambiar sus valores, lo cual es poco probable que suceda justo al mismo tiempo. En un circuito asíncrono los valores de las variables de estado siguiente están determinados por redes de compuertas



a) Diagrama de estado

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	A	(D)	0

b) Tabla de flujo

Figura 9.13 FSM asíncrona que genera paridad.

lógicas con retrasos de propagación que varían. En consecuencia, debemos esperar que una variable de estado cambie ligeramente antes que la otra, lo cual pondría al circuito en un estado donde podría reaccionar a la entrada de una manera no deseada. Suponga que y_1 cambia primero. Entonces el circuito pasa de $y_2y_1 = 11$ a $y_2y_1 = 10$. En cuanto alcanza este estado, *C*, intentará permanecer ahí si $w = 0$, lo cual es un resultado erróneo. Por otro lado, suponga que y_2 cambia primero. Entonces habrá un cambio de $y_2y_1 = 11$ a $y_2y_1 = 01$, que corresponde al estado *B*. Como $w = 0$, el circuito ahora tratará de cambiar a $y_2y_1 = 10$. De nuevo, esto requiere que tanto y_1 como y_2 cambien; suponiendo que y_1 cambie primero en la transición de $y_2y_1 = 01$, el circuito se encontrará en el estado $y_2y_1 = 00$, que es el estado destino correcto, *A*. Esta explicación indica que la transición requerida de *D* a *A* se realizará correctamente si y_2 cambia antes que y_1, pero no funcionará si y_1 cambia antes que y_2. El resultado depende de la "carrera" para cambiar entre las señales y_1 y y_2.

La incertidumbre causada por los múltiples cambios en las variables de estado en respuesta a una entrada que debe conducir a un cambio predecible de un estado estable a otro tiene que eliminarse. El término *condición de carrera* se usa para referirse a este comportamiento impredecible. Estudiaremos este tema con profundidad en la sección 9.5.

Estado presente y_2y_1	Estado siguiente Y_2Y_1 $w = 0$	$w = 1$	Salida z
00	(00)	01	0
01	10	(01)	1
10	(10)	11	1
11	00	(11)	0

a) Mala asignación de estados

Estado presente y_2y_1	Estado siguiente Y_2Y_1 $w = 0$	$w = 1$	Salida z
00	(00)	01	0
01	11	(01)	1
11	(11)	10	1
10	00	(10)	0

b) Buena asignación de estados

Figura 9.14 Asignación de estados para la figura 9.13*b*.

Las condiciones de carrera pueden eliminarse tratando las variables de estado presente como si fueran entradas al circuito, lo que significa que sólo se permite el cambio de una variable de estado a la vez. Para nuestro ejemplo, la asignación $A = 00$, $B = 01$, $C = 11$ y $D = 10$ logra este objetivo. La tabla de excitación resultante se presenta en la figura 9.14*b*. El lector debe comprobar que todas las transiciones suponen el cambio de una sola variable de estado.

A partir de la figura 9.14*b* las expresiones de estado siguiente y de salida son

$$Y_1 = w\bar{y}_2 + \bar{w}y_1 + y_1\bar{y}_2$$
$$Y_2 = wy_2 + \bar{w}y_1 + y_1y_2$$
$$z = y_1$$

El último término producto en las expresiones para Y_1 y Y_2 se incluye para cubrir todos los riesgos posibles, los cuales se estudian en la sección 9.6. El circuito correspondiente se muestra en la figura 9.15.

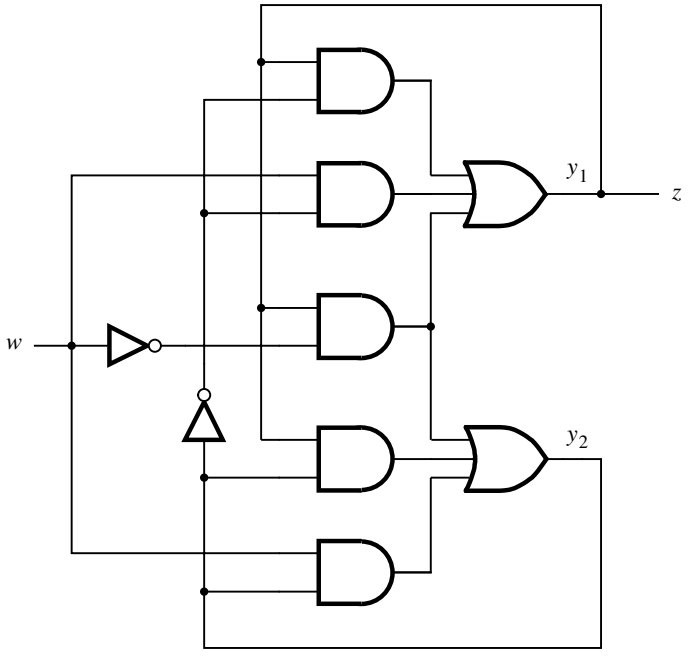


Figura 9.15 Circuito que implementa la FSM de la figura 9.13*b*.

Es interesante considerar cómo el generador de paridad serial podría implementarse adoptando un enfoque síncrono. Todo lo que se necesita es un flip-flop individual que cambie su estado con la llegada de cada pulso de entrada. El flip-flop D disparado por flanco positivo de la figura 9.16 realiza esta tarea, suponiendo que se establece inicialmente en Q = 0. La complejidad lógica del flip-flop es exactamente la misma que la del circuito de la figura 9.15. De hecho, si usamos las expresiones anteriores para Y_1 y Y_2 y sustituimos C por w, D por $\overline{y}_2$, y_m por y_1 y y_s por y_2, terminamos con las expresiones de excitación del flip-flop D maestro-esclavo del ejemplo 9.2. El circuito de la figura 9.15 es en realidad un flip-flop maestro-esclavo disparado por flanco negativo, con el complemento de su salida Q (y_2) conectada a su entrada D. La salida z está conectada a la salida de la etapa de maestro del flip-flop.

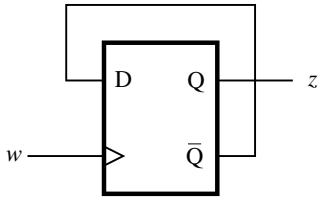


Figura 9.16 Solución síncrona para el ejemplo 9.4.

Ejemplo 9.5

CONTADOR MÓDULO 4 En los capítulos 7 y 8 describimos cómo pueden implementarse los contadores utilizando flip-flops. Ahora sintetizaremos un contador como un circuito secuencial asíncrono. En la figura 9.17 se representa un diagrama de estado para un contador módulo 4, el cual cuenta el número de pulsos en una línea de entrada, w. El circuito debe ser capaz de reaccionar a todos los cambios en la señal de entrada; por tanto, ha de tomar medidas específicas tanto en los flancos positivos como en los negativos de cada pulso. Por consiguiente, se precisan ocho estados para ocuparse de los flancos en cuatro pulsos consecutivos.

El contador comienza en el estado A y permanece en él mientras $w = 0$. Cuando w cambia a 1 se lleva a cabo una transición al estado B y el circuito se mantiene estable en este estado mientras $w = 1$. Cuando w regresa a 0, el circuito pasa al estado C y permanece estable hasta que w se vuelve 1 otra vez, lo que ocasiona una transición al estado D, y así sucesivamente. Al usar el modelo Moore, los estados corresponden a conteos específicos. Hay dos estados para cada conteo en particular: el estado en el que entra la FSM cuando w cambia de 0 a 1 al principio de un pulso y el estado en el que la FSM entra cuando w regresa a 0 al final del pulso. Los estados B y C corresponden al conteo de 1, los estados D y E a 2 y los estados F y G a 3. Los estados A y H representan el conteo de 0.

En la figura 9.18 se muestran las tablas de flujo y de excitación para el contador. La asignación de estados se elige de tal modo que todas las transiciones entre estados requieran el cambio de valor de sólo una variable de estado para eliminar la posibilidad de las condiciones de carrera. La salida se codifica como un número binario, usando las variables z_2 y z_1. A partir de la tabla de excitación las expresiones de estado siguiente y de salida son

$$\begin{aligned} Y_1 &= \overline{w}y_1 + wy_2y_3 + w\overline{y}_2\overline{y}_3 + y_1y_2y_3 + y_1\overline{y}_2\overline{y}_3 \\ &= \overline{w}y_1 + (w + y_1)(y_2y_3 + \overline{y}_2\overline{y}_3) \\ Y_2 &= wy_2 + \overline{w}y_1\overline{y}_3 + \overline{y}_1y_2 + y_2\overline{y}_3 \end{aligned}$$

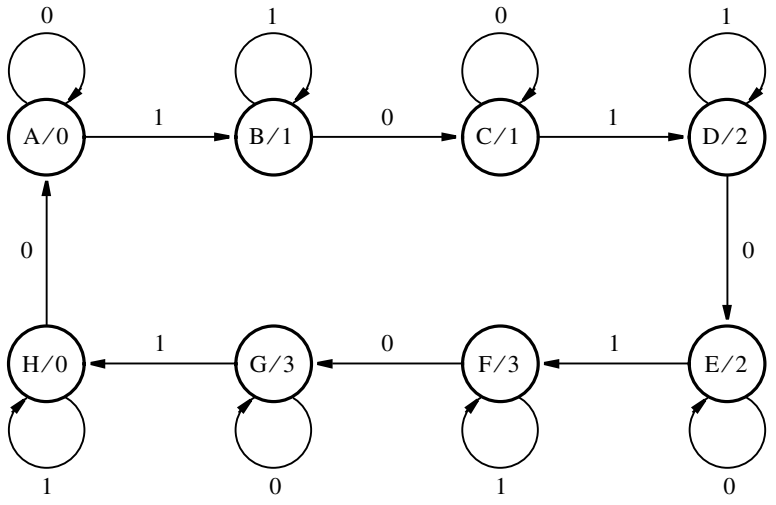


Figura 9.17 Diagrama de estado para un contador módulo 4.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	E	(D)	2
E	(E)	F	2
F	G	(F)	3
G	(G)	H	3
H	A	(H)	0

a) Tabla de flujo

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	
$y_3y_2y_1$	$Y_3Y_2Y_1$		z_2z_1
000	(000)	001	00
001	011	(001)	01
011	(011)	010	01
010	110	(010)	10
110	(110)	111	10
111	101	(111)	11
101	(101)	100	11
100	000	(100)	00

b) Tabla de excitación

Salida módulo 8
$z_3z_2z_1$
000
001
010
011
100
101
110
111

c) Salida para el conteo de los flancos

Figura 9.18 Tablas de flujo y de excitación para un contador módulo 4.

$$Y_3 = wy_3 + y_1y_3 + \overline{y}_1y_2\overline{w} + y_2y_3$$
$$z_1 = y_1$$
$$z_2 = y_1y_3 + \overline{y}_1y_2$$

Estas expresiones definen el circuito que implementa el contador de pulsos módulo 4 requerido.

En la derivación anterior diseñamos un circuito que cambia su estado en cada flanco de la señal de entrada *w*, para el que se precisaba un total de ocho estados. Como se supone que el circuito cuenta el número de pulsos completos, el cual contiene un flanco en ascenso y uno en descenso, el conteo de salida z_2z_1 cambia su valor sólo en cada segundo estado. Esta FSM se comporta como un circuito secuencial síncrono en el que el conteo de salida cambia sólo como consecuencia de que *w* cambie de 0 a 1.

Suponga ahora que se quiere contar el número de veces que la señal *w* cambia su valor, es decir, el número de sus flancos. Las transiciones de estado que se especifican en las figuras 9.17 y 9.18 definen una FSM que puede operar como un contador módulo 8 para este propósito. Sólo debemos especificar una salida distinta en cada estado, lo que puede hacerse como se muestra en la figura 9.18*c*. Los valores de $z_3z_2z_1$ indican la secuencia de conteo 0, 1, 2,..., 7, 0. Usando esta especificación para la salida y la asignación de estados de la figura 9.18*b*, las expresiones de salida resultantes son

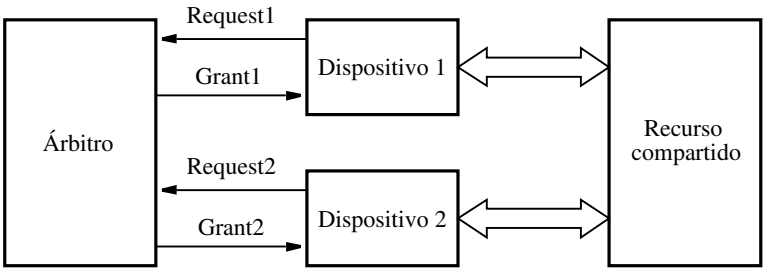
$$z_1 = y_1 \oplus y_2 \oplus y_3$$
$$z_2 = y_2 \oplus y_3$$
$$z_3 = y_3$$

Ejemplo 9.6

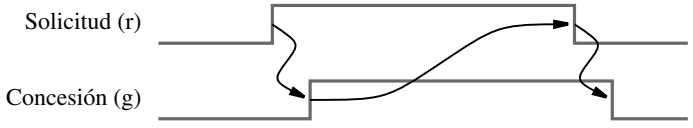
UN ÁRBITRO SIMPLE En los sistemas de cómputo con frecuencia es útil tener algún recurso compartido por varios dispositivos. En general, el recurso puede usarlo sólo un dispositivo a la vez. Cuando varios dispositivos necesitan utilizarlo deben solicitarlo. Estas solicitudes son manejadas por un circuito árbitro. Cuando existen dos o más solicitudes simultáneas, el árbitro puede usar algún esquema de prioridad para elegir una de ellas, como ya vimos en la sección 8.8.

Ahora consideraremos un ejemplo de un árbitro simple implementado como un circuito secuencial asíncrono. Para mantener el ejemplo pequeño, suponga que dos dispositivos están compitiendo por el recurso compartido, como se indica en la figura 9.19*a*. Cada dispositivo se comunica con el árbitro por medio de dos señales: *Request (solicitud)* y *Grant (concesión)*. Cuando un dispositivo necesita usar el recurso compartido, activa su señal Request a 1. Luego espera hasta que el árbitro responde con la señal Grant.

En la figura 9.19*b* se ilustra un esquema de uso común para la comunicación entre dos entidades en el entorno asíncrono, conocido como *señalización de reconocimiento* (*handshake signaling*). Dos señales se usan para proporcionar el reconocimiento. Un dispositivo inicia la actividad al hacer una solicitud, $r = 1$. Cuando el recurso compartido está disponible, el árbitro responde emitiendo una concesión, $g = 1$. Cuando el dispositivo recibe la señal de concesión, procede a usar el recurso compartido solicitado. Cuando termina de emplearlo, retira su solicitud al establecer $r = 0$. Cuando el árbitro ve que $r = 0$, desactiva la señal de concesión, haciendo $g = 0$. Las flechas de la figura indican las relaciones de causa-efecto de este esquema de señali-



a) Estructura del árbitro



b) Señalización de reconocimiento

Figura 9.19 Ejemplo de árbitro.

zación; un cambio en una de las señales provoca un cambio en la otra señal. El tiempo transcurrido entre los cambios en las señales de causa-efecto depende de la implementación específica del circuito. Un punto clave es que no hay necesidad de un reloj de sincronización.

En la figura 9.20 se presenta un diagrama de estado para nuestro árbitro simple. Hay dos entradas, las señales de solicitud r_1 y r_2, y dos salidas, las señales de concesión g_1 y g_2. El diagrama describe el modelo Moore de la FSM requerida, donde los arcos se etiquetan como r_2r_1 y las salidas de estado como g_2g_1. El estado inactivo es A, donde no hay solicitudes. El estado B representa la situación en la que el Dispositivo 1 recibe permiso para usar el recurso, y el estado C indica lo mismo para el Dispositivo 2. Por tanto, B es estable si $r_2r_1 = 01$, y C lo es si $r_2r_1 = 10$. Para apegarse a las reglas del diseño de circuitos asíncronos, supondremos que las entradas r_1 y r_2 se activarán una a la vez. Por consiguiente, en el estado A es imposible tener un cambio de $r_2r_1 = 00$ a $r_2r_1 = 11$. La situación donde $r_2r_1 = 11$ ocurre sólo cuando se hace una segunda solicitud antes que el dispositivo que tiene la señal de concesión complete su uso del recurso compartido, lo cual sucede en los estados B y C. Si la FSM es estable ya sea en el estado B o C, permanecerá en este estado si tanto r_1 como r_2 pasan a 1.

La tabla de flujo se proporciona en la figura 9.21*a* y la de excitación en la 9.21*b*. Es imposible elegir una asignación de estado en la que todos los cambios entre los estados A, B y C impliquen un cambio únicamente en una sola variable de estado. En la asignación elegida las transiciones hacia o desde el estado A se manejan de manera apropiada, pero las transiciones entre los estados B y C suponen cambios en los valores de las dos variables de estado y_1 y y_2. Supóngase que el circuito se halla estable en el estado B bajo la combinación de la entrada $r_2r_1 = 11$. Ahora digamos que las entradas cambian a $r_2r_1 = 10$. Esto debe causar un cambio hacia el estado C,

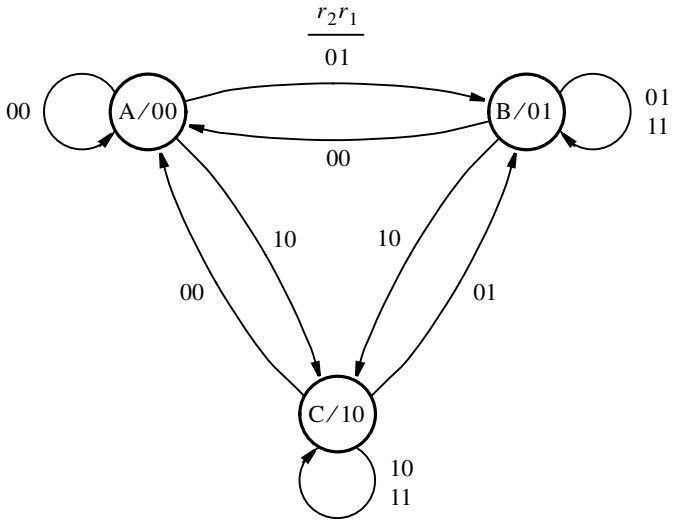


Figura 9.20 Diagrama de estado para el árbitro.

Estado presente	Estado siguiente				Salida
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(A)	B	C	–	00
B	A	(B)	C	(B)	01
C	A	B	(C)	(C)	10

a) Tabla de flujo

	Estado presente	Estado siguiente				Salida
		$r_2r_1 = 00$	01	10	11	
	y_2y_1	Y_2Y_1				g_2g_1
A	00	(00)	01	10	–	00
B	01	00	(01)	10	(01)	01
C	10	00	01	(10)	(10)	10
D	11	–	01	10	–	*dd*

b) Tabla de excitación

Figura 9.21 Implementación del árbitro.

lo que significa que las variables de estado deben cambiar de $y_2 y_1 = 01$ a 10. Si y_1 cambia más rápido que y_2, entonces el circuito se encontrará de manera momentánea en el estado $y_2 y_1 = 00$, lo que conduce al estado final buscado porque desde el estado *A* hay una transición especificada a *C* bajo la combinación de entrada 10. Pero si y_2 cambia más rápido que y_1, el circuito alcanzará el estado $y_2 y_1 = 11$, lo cual no está definido en la tabla de flujo. Para asegurar que aun en este caso el circuito procederá al destino *C* requerido, podemos incluir el estado $y_2 y_1 = 11$, etiquetado *D*, en la tabla de excitación y especificar la transición requerida como se muestra en la figura. Una situación parecida surge cuando el circuito está estable en *C* bajo $r_2 r_1 = 11$, y debe cambiar a *B* cuando r_2 cambia de 1 a 0.

Los valores de salida para el estado adicional *D* se indican como condiciones no-importa. Siempre que una salida específica está cambiando de 0 a 1 o de 1 a 0, no es importante exactamente cuando este cambio ocurre si el valor correcto se produce cuando el circuito se halla en un estado estable. La especificación no-importa puede conducir a una realización más simple de las funciones de salida. Es importante asegurar que las salidas sin especificar no darán como resultado un valor que pueda causar un comportamiento erróneo. Con base en la figura 9.21*b* es posible que durante el breve tiempo que precisa el circuito para pasar por el estado inestable *D* las salidas se vuelven $g_2 g_1 = 11$. Esto no es perjudicial en nuestro ejemplo porque el dispositivo que acaba de usar el recurso compartido no tratará de utilizarlo de nuevo hasta que su señal de concesión ha regresado a 0 para indicar el fin del reconocimiento con el árbitro. Obsérvese que si esta condición ocurre durante un cambio de *B* a *C*, entonces g_1 aún es 1 poco tiempo después y g_2 se vuelve 1 poco tiempo antes. De manera similar, si la transición va de *C* a *B* entonces el cambio en g_1 de 0 a 1 ocurre poco tiempo antes y g_2 cambia a 0 poco tiempo después. En ambos casos no hay un mal funcionamiento ni en g_1 ni en g_2.

A partir de la tabla de excitación se derivan las expresiones de estado siguiente y de salida

$$\begin{aligned} Y_1 &= \bar{r}_2 r_1 + r_1 \bar{y}_2 \\ Y_2 &= r_2 \bar{r}_1 + r_2 y_2 \\ g_1 &= y_1 \\ g_2 &= y_2 \end{aligned}$$

Al reescribir las primeras dos expresiones como

$$\begin{aligned} Y_1 &= r_1(\bar{r}_2 + \bar{y}_2) \\ &= r_1 \overline{r_2 y_2} \\ Y_2 &= r_2(\bar{r}_1 + y_2) \end{aligned}$$

se produce el circuito de la figura 9.22. Nótese que este circuito responde muy rápidamente a los cambios en las señales de entrada. Este comportamiento presenta un marcado contraste con el árbitro estudiado en la sección 8.8, en el que el reloj de sincronización determina el tiempo de respuesta mínimo.

La dificultad con la condición de carrera que surge por los cambios de estado entre *B* y *C* puede resolverse de otra manera. Simplemente podemos evitar que el circuito llegue a un estado no especificado. En la figura 9.23*a* se muestra una tabla de flujo modificada en la que las transiciones entre los estados *B* y *C* se realizan mediante el estado *A*. Si el circuito está estable en *B* y la combinación de entrada cambia de $r_2 r_1 = 11$ a 10, primero ocurrirá un cambio a *A*. En cuanto el circuito entra en el estado *A*, que no es estable para la combinación de entrada 10, proseguirá hacia el estado estable *C*. El desvío a través del estado inestable *A* es aceptable porque en tal estado la salida es $g_2 g_1 = 00$, lo cual es consistente con la operación deseada del árbitro. El cambio

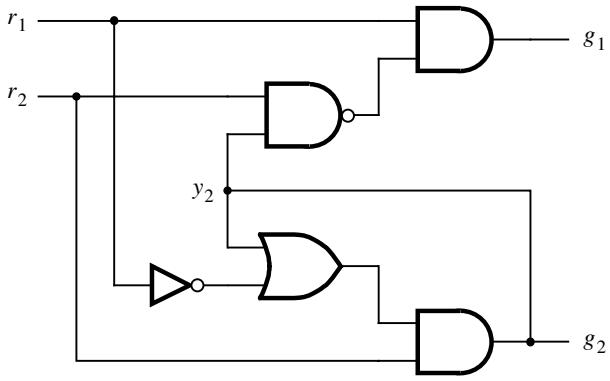


Figura 9.22 El circuito árbitro.

Estado presente	Estado siguiente				Salida
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(A)	B	C	–	00
B	A	(B)	A	(B)	01
C	A	A	(C)	(C)	10

a) Tabla de flujo modificada

Estado presente	Estado siguiente				Salida
	$r_2r_1 = 00$	01	10	11	
y_2y_1	Y_2Y_1				g_2g_1
00	(00)	01	10	–	00
01	00	(01)	00	(01)	01
10	00	00	(10)	(10)	10

b) Tabla de excitación modificada

Figura 9.23 Una alternativa para evitar una carrera crítica en la figura 9.21.

de *C* a *B* se maneja usando el mismo enfoque. A partir de la tabla de excitación modificada de la figura 9.23*b* se derivan estas expresiones de estado siguiente:

$$Y_1 = r_1\bar{y}_2$$
$$Y_2 = \bar{r}_1 r_2 \bar{y}_1 + r_2 y_2$$

Estas expresiones dan origen a un circuito distinto al de la figura 9.22. Sin embargo, los dos circuitos implementan la funcionalidad requerida en el árbitro.

A continuación intentaremos diseñar el mismo árbitro utilizando la especificación del modelo Mealy. Con base en la figura 9.20 es evidente que los estados *B* y *C* son fundamentalmente distintos porque para la entrada $r_2r_1 = 11$ deben producir dos salidas distintas. Pero el estado *A* es único sólo en la medida en que genera la salida $g_2g_1 = 00$ siempre que $r_2r_1 = 00$. Esta condición podría especificarse tanto en *B* como en *C* si se emplea el modelo Mealy. En la figura 9.24 se presenta un diagrama de estado adecuado. Las tablas de flujo y de excitación se muestran en la figura 9.25, y conducen a las expresiones siguientes

$$Y = r_2\bar{r}_1 + \bar{r}_1 y + r_2 y$$
$$g_1 = r_1\bar{y}$$
$$g_2 = r_2 y$$

Pese a necesitar una sola variable de estado, la implementación de este circuito precisa más compuertas que la versión Moore de la figura 9.22.

Una noción importante en el ejemplo anterior consiste en que es necesario prestar mucha atención a la asignación de estados para evitar carreras cuando los valores de las variables de estado cambien. En la sección 9.5 abordaremos este aspecto con más detalle.

Hicimos la suposición básica de que las entradas de solicitud a la FSM árbitro cambian sus valores uno a la vez, lo que permite al circuito alcanzar un estado estable antes que el siguiente cambio ocurra. Si los dispositivos son totalmente independientes pueden hacer sus solicitudes en cualquier momento. Supóngase que cada uno de ellos realiza una solicitud cada pocos segundos. Puesto que el circuito árbitro necesita sólo unos cuantos nanosegundos para cambiar de un estado estable a otro, es muy poco probable que ambos dispositivos hagan sus solicitudes tan cercanas unas de otras que ocasionen que el circuito árbitro produzca salidas erróneas. No obstante, si bien la probabilidad de un error causado por la llegada simultánea de las solicitudes es muy baja, no es cero. Si esta pequeña posibilidad de error no puede tolerarse, entonces es posible alimentar las señales de la solicitud por medio de un circuito especial llamado elemento de *exclusión mutua* (ME, *mutual exclusion*). Este circuito tiene dos entradas y dos salidas. Si las dos entradas son 0, entonces las dos salidas son 0. Si sólo una entrada es 1, entonces la salida

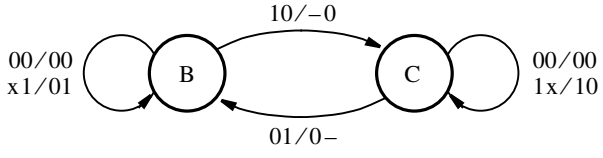


Figura 9.24 Modelo Mealy para la FSM árbitro.

Estado presente	Estado siguiente				Salida g_2g_1			
	$r_2r_1 = 00$	01	10	11	00	01	10	11
B	(B)	(B)	C	(B)	00	01	−0	01
C	(C)	B	(C)	(C)	00	0−	10	10

a) Tabla de flujo

Estado presente y	Estado siguiente				Salida			
	$r_2r_1 = 00$	01	10	11	00	01	10	11
	Y				g_2g_1			
0	(0)	(0)	1	(0)	00	01	$d0$	01
1	(1)	0	(1)	(1)	00	$0d$	10	10

b) Tabla de excitación

Figura 9.25 Implementación del modelo Mealy de la FSM árbitro.

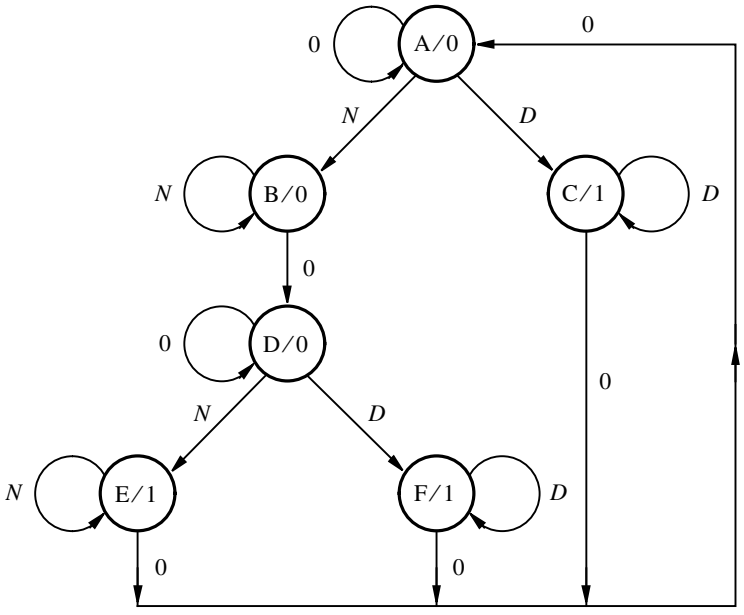
correspondiente es 1. Si las dos entradas son 1, el circuito hace que una salida vaya a 1 y mantiene la otra en 0. Si se ocupa el elemento ME cambiaría el diseño del árbitro ligeramente; como la combinación $r_2r_1 = 11$ nunca ocurriría, todas las entradas en la columna correspondiente de la figura 9.21 serían condiciones no-importa. El elemento ME y la cuestión de los cambios simultáneos en las señales de entrada se estudian con detalle en la referencia [6]. Finalmente, cabe señalar que un problema semejante surge en los circuitos síncronos en los cuales una o más entradas se generan mediante un circuito que no está controlado por un reloj común. Abordaremos este tema en la sección 10.3.3.

9.4 REDUCCIÓN DE ESTADOS

En el capítulo 8 vimos que reducir el número de estados necesarios para producir la funcionalidad de una FSM suele conducir a menos variables de estado, lo cual significa que se requieren menos flip-flops en el circuito secuencial síncrono correspondiente. En los circuitos secuenciales asíncronos también es útil tratar de reducir el número de estados porque ello casi siempre da como resultado implementaciones más simples.

Cuando se diseña una FSM asíncrona, es probable que la tabla de flujo inicial tenga muchas entradas sin especificar (no-importa), ya que el diseñador tiene que obedecer la restricción de que sólo una variable de entrada a la vez puede cambiar su valor. Supóngase que queremos diseñar la FSM para la máquina expendedora simple considerada en el ejemplo 9.3. Recuérdese que la máquina acepta monedas de 5 y 10 centavos y entrega caramelos cuando se depositan 10 centavos; la máquina no da cambio si se depositan 15 centavos. Un diagrama de estado inicial

para esta FSM puede derivarse de manera sencilla al enumerar todas las secuencias posibles de depositar las monedas para dar una suma de cuando menos 10 centavos. En la figura 9.26*a* se muestra un diagrama posible, definido como un modelo Moore. Comenzando en un estado reset, *A*, la FSM permanece ahí mientas no se deposite ninguna moneda, lo cual se indica por medio de un arco etiquetado 0 para indicar que $N = D = 0$. Ahora se coloca un arco con la etiqueta *N* que



a) Diagrama de estado inicial

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	F	–	0
E	A	(E)	–	–	1
F	A	–	(F)	–	1

b) Tabla de flujo inicial

Figura 9.26 Derivación de una FSM para la máquina expendedora simple.

indica que el mecanismo de detección de monedas ha detectado una moneda de cinco centavos y ha generado una señal $N = 1$. De forma similar, sea D la indicación de que se depositó una moneda de 10 centavos. Si $N = 1$, entonces la FSM debe moverse a un estado nuevo, digamos, B, y debe permanecer estable en él siempre que N tenga el valor de 1. Como B corresponde al hecho de que se han depositado cinco centavos, la salida de este estado debe ser 0. Si se depositó una moneda de 10 centavos en el estado A, entonces la FSM debe pasar a un estado distinto, digamos, C. La máquina debe mantenerse en C mientras $D = 1$, y debe entregar el caramelo al generar la salida de 1. Éstas son las únicas transiciones factibles del estado A, puesto que es imposible insertar dos monedas al mismo tiempo, lo que significa que $DN = 11$ puede tratarse como una condición no-importa. Enseguida, en el estado B debe haber una vuelta a la condición $DN = 00$ porque el mecanismo detector de monedas detectará la segunda moneda tiempo después que la primera borra el mecanismo. Este comportamiento es consistente con el requisito de que sólo una variable de entrada a la vez puede cambiar; por consiguiente no está permitido ir de $DN = 01$ a $DN = 10$. La entrada $DN = 10$ no puede ocurrir en el estado B y debe tratarse como una condición no-importa. La entrada $DN = 00$ lleva a la FSM a un estado nuevo, D, el cual indica que se depositaron cinco centavos y que no hay una moneda en el mecanismo detector. En el estado D es posible depositar una moneda de cinco centavos o una de 10. Si $DN = 01$, la máquina pasa al estado E, que indica que se depositaron 10 centavos y genera la salida de 1. Si $DN = 10$, la máquina pasa al estado F, el cual también genera la salida de 1. Finalmente, cuando la FSM está en cualquiera de los estados C, E o F, la única entrada posible es $DN = 00$, la cual regresa la máquina al estado A.

La tabla de flujo para esta FSM se presenta en la figura 9.26*b*. Muestra de manera explícita todas las entradas no-importa. Estas entradas sin especificar brindan cierta flexibilidad que puede aprovecharse al reducir el número de estados. Obsérvese que en cada fila de esta tabla hay únicamente un estado estable. Tales tablas, donde sólo hay un estado estable por cada fila, se conocen como *tablas de flujo primitivas*.

Se han desarrollado varias técnicas para la reducción de estados. En esta sección describiremos un proceso de dos pasos. En el primer paso aplicaremos el procedimiento de particionamiento de la sección 8.6.1, suponiendo que las filas potencialmente equivalentes en una tabla de flujo deben producir las mismas salidas. Como restricción adicional, para que dos filas sean potencialmente equivalentes cualesquiera entradas sin especificar deben estar en las mismas columnas del estado siguiente. De esta manera, al combinar los estados equivalentes en un solo estado no se eliminarán las entradas no-importa ni la flexibilidad que ofrecen. En el segundo paso, las filas se *fusionan* al aprovechar las entradas sin especificar. Dos filas pueden fusionarse si no tienen entradas de estado siguiente en conflicto. Esto significa que sus valores de estado siguiente para una combinación dada de entradas son iguales, una de ellas está sin especificar o las dos filas indican un estado estable. Si se usa el modelo Moore, entonces las dos filas (estados) deben producir las mismas salidas. Si se recurre al modelo Mealy, entonces los dos estados deben producir las mismas salidas para cualesquiera combinaciones para las que ambos estados son estables.

Ejemplo 9.7

Ahora mostraremos cómo el diagrama de flujo de la figura 9.26*b* puede reducirse a la forma mejorada de la figura 9.12. El primer paso en el proceso de reducción de estados es el procedimiento de particionamiento de la sección 8.6.1. Los estados A y D son estables bajo la combinación de entrada $DN = 00$, lo que produce la salida de 0; también tienen entradas sin especificar en la misma posición. Los estados C y F son estables bajo $DN = 10$, lo que genera $z = 1$, y también tienen las mismas entradas sin especificar. Los estados B y E tienen las mismas entradas sin especificar, pero cuando son estables bajo $DN = 01$ el estado B produce $z = 0$, mientras E genera $z = 1$; no son

equivalentes. Por consiguiente, la partición inicial es

$$P_1 = (AD)(B)(CF)(E)$$

Los sucesores de A y D son (A, D) para $DN = 00$, (B, E) para 01 y (C, F) para 10. Como el par (B, E) no está en el mismo bloque de P_1, se deduce que los estados A y D no son equivalentes. Los sucesores de C y F son (A, A) para 00 y (C, F) para 10; cada par está en un solo bloque. Por tanto, la segunda partición es

$$P_2 = (A)(D)(B)(CF)(E)$$

Los sucesores de C y F en P_1 están en el mismo bloque de P_2, lo cual significa que

$$P_3 = P_2$$

La conclusión es que las filas C y F son equivalentes. Al combinarlas en una sola fila y cambiar todas las F por C se obtiene la tabla de flujo de la figura 9.27.

A continuación podemos tratar de fusionar algunas filas en la tabla de flujo aprovechando la existencia de entradas sin especificar. La única fila que puede fusionarse con otras es C. Puede fusionarse con A o con E, pero no con las dos. Fusionar C con A significaría que el nuevo estado debe generar $z = 0$ cuando está estable bajo la combinación de entrada 00 y ha de producir $z = 1$ cuando se encuentra estable bajo 10. Esto sólo puede lograrse si se utiliza el modelo Mealy. La alternativa es fusionar C y E, caso en el que el nuevo estado se halla estable bajo $DN = 01$ y 10, lo que produce la salida de 1. Esto puede lograrse con el modelo Moore. Fusionar C y E en un solo estado C y cambiar todas las E por C genera la tabla de flujo reducida de la figura 9.12. Observe que cuando C y E se fusionan, la nueva fila C debe incluir todas las especificaciones de ambas filas C y E. Las dos filas especifican a A como el estado siguiente si $DN = 00$. La fila E especifica un estado estable para $DN = 01$; por consiguiente la nueva fila (llamada C) también debe especificar un estado estable para la misma combinación. De manera similar, la fila C especifica un estado estable para $DN = 10$, lo cual debe reflejarse en la nueva fila. Por ende, las entradas del estado siguiente en la fila nueva son A, (C) y (C) para las combinaciones de entrada 00, 01 y 10, respectivamente.

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	C	–	0
E	A	(E)	–	–	1

Figura 9.27 Primer paso de reducción de la FSM de la figura 9.26*b*.

Procedimiento de fusión

En el ejemplo 9.7 fue fácil decidir qué filas deben fusionarse porque las únicas posibilidades son fusionar la fila *C* ya sea con *A* o con *E*. Elegimos fusionar *C* y *E* debido a que esto puede hacerse preservando el modelo Moore, el cual es probable que nos conduzca a una expresión más simple que produce la salida *z*.

En general, puede haber muchas posibilidades para fusionar filas en tablas de flujo más grandes. En estos casos es preciso contar con un procedimiento más estructurado para hacer la elección. Un procedimiento útil puede definirse usando el concepto de *compatibilidad* de estados.

Definición 9.1 *Se dice que dos estados (filas en una tabla de flujo), S_i y S_j, son compatibles si no existen conflictos de estado para ninguna combinación de entrada. Por tanto, para cada combinación de entrada una de las condiciones siguientes debe ser verdadera:*

- *tanto S_i como S_j tienen el mismo sucesor, o*
- *tanto S_i como S_j son estables, o*
- *el sucesor de S_i o S_j, o ambos, no está especificado.*

Además, tanto S_i como S_j deben tener la misma salida siempre que se especifique.

Considérese la tabla de flujo primitiva de la figura 9.28. Examinemos la compatibilidad entre los diferentes estados, suponiendo que quisiéramos conservar la especificación del tipo Moore de salidas para esta FSM. El estado *A* es compatible sólo con el estado *H*. El estado *B* es compatible con los estados *F* y *G*. El estado *C* no es compatible con ningún otro estado. El estado *D* es compatible con el estado *E*, el estado *F* con *G*, y el estado *G* con *H*. En otras palabras, existen los siguientes pares compatibles: (*A*, *H*), (*B*, *F*), (*B*, *G*), (*D*, *E*), (*F*, *G*) y (*G*, *H*). La relación de compatibilidad entre varios estados puede representarse de manera práctica en la forma de un *diagrama de fusión*, como sigue:

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	H	B	–	0
B	F	–	(B)	C	0
C	–	H	–	(C)	1
D	A	(D)	–	E	1
E	–	D	G	(E)	1
F	(F)	D	–	–	0
G	F	–	(G)	–	0
H	–	(H)	–	E	0

Figura 9.28 Una tabla de flujo primitiva.

- Cada fila de la tabla de flujo se representa como un punto, etiquetado con el nombre de la fila.
- Se traza una línea que conecte cualesquiera dos puntos que corresponden a los estados compatibles (filas).

A partir del diagrama de fusión puede elegirse la mejor posibilidad de fusión y derivarse la tabla de flujo reducida.

En la figura 9.29 se observa el diagrama de fusión para la tabla de flujo primitiva de la figura 9.28. El diagrama indica que la fila *A* puede fusionarse con *H*, pero sólo si *H* no se fusiona con *G*, pues no hay una línea que una *A* y *G*. La fila *B* puede fusionarse con las filas *F* y *G*. Como también es posible fusionar *F* y *G*, se deduce que *B*, *F* y *G* son compatibles con todos los pares posibles del conjunto. Cualquier conjunto de filas que sean compatibles con todos los pares del conjunto puede fusionarse en un solo estado. Por tanto, los estados *B*, *F* y *G* pueden fusionarse en un solo estado, pero sólo si los estados *G* y *H* no están fusionados. El estado *C* no puede fusionarse con ningún otro estado. Los estados *D* y *E* pueden fusionarse.

Una estrategia prudente consiste en fusionar los estados de modo que la tabla de flujo resultante tenga tan pocos estados como sea posible. En nuestro ejemplo la mejor opción es fusionar los compatibles (*A*, *H*), (*B*, *F*, *G*) y (*D*, *E*), lo cual conduce a la tabla de flujo reducida de la figura 9.30.

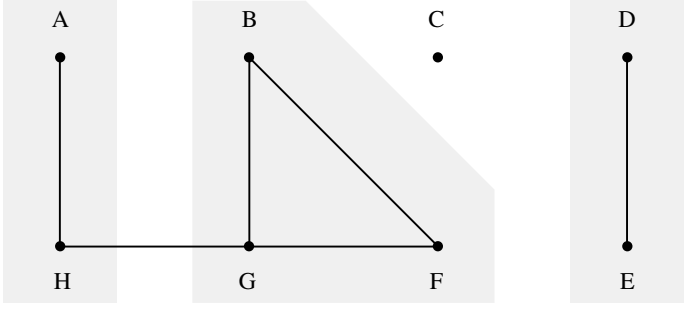


Figura 9.29 Diagrama de fusión para la tabla de flujo de la figura 9.28, que conserva el modelo Moore.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	B	D	0
B	(B)	D	(B)	C	0
C	–	A	–	(C)	1
D	A	(D)	B	(D)	1

Figura 9.30 Tabla de flujo tipo Moore reducida para la FSM de la figura 9.28.

Cuando se crea una fila nueva al fusionar dos o más filas, todas las entradas en la nueva fila deben especificarse para cubrir los requisitos individuales de las filas constitutivas. El remplazo de las filas *A* y *H* con una nueva fila *A* requiere volver *A* estable tanto para $w_2w_1 = 00$ como para 01, ya que el viejo estado *A* debe ser estable para 00 y *H* ha de serlo para 01. También requiere especificar *B* como el estado siguiente para $w_2w_1 = 10$ y *E* como el estado siguiente para $w_2w_1 = 11$. Como el viejo estado *E* se vuelve *D*, después de fusionar *D* y *E*, la nueva fila *A* debe tener las entradas del estado siguiente (*A*), (*A*), *B* y *D* para las combinaciones de entrada 00, 01, 10 y 11, respectivamente. Remplazar las filas *B*, *F* y *G* con una nueva fila *B* requiere volver a *B* estable para $w_2w_1 = 00$ y 10. La entrada de estado siguiente para $w_2w_1 = 01$ debe ser *D* para satisfacer el requisito del viejo estado *F*. La entrada de estado siguiente para $w_2w_1 = 11$ tiene que ser *C*, según dicta el viejo estado *B*. Obsérvese que el viejo estado *G* no impone requisitos para las transiciones bajo $w_2w_1 = 01$ y 11, pues sus entradas del estado siguiente correspondientes no están especificadas. La fila *C* aún es igual que antes excepto que el nombre de la entrada del estado siguiente para $w_2w_1 = 01$ debe cambiarse de *H* a *A*. Las filas *D* y *E* son reemplazadas por una fila nueva *D*, usando un razonamiento parecido. Nótese que la tabla de flujo de la figura 9.30 aún es del tipo Moore.

Hasta ahora consideramos fusionar sólo las filas que nos permitirían conservar la especificación del tipo Moore de la FSM de la figura 9.28. Si estamos dispuestos a cambiar al modelo Mealy, entonces hay otras posibilidades. En la figura 9.31 se muestra el diagrama de fusión completo para la FSM de la figura 9.28. Las líneas negras conectan los estados compatibles que pueden fusionarse en un estado nuevo que tiene una salida tipo Moore; esto corresponde al diagrama de fusión de la figura 9.29. Las líneas grises conectan los estados que pueden fusionarse sólo si se usan las salidas tipo Mealy.

En este caso es poco probable que recurrir al modelo Mealy produzca un circuito más simple. Aun cuando existen varias posibilidades de fusión, todas requieren cuando menos cuatro estados en la tabla de flujo reducida, lo que no es nada mejor que la solución obtenida en la figura 9.30. Por ejemplo, una posibilidad es realizar la fusión con base en la partición (*A*, *H*), (*B*, *C*, *G*)

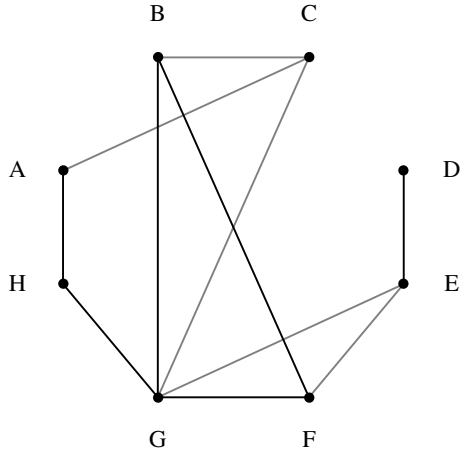


Figura 9.31 Diagrama de fusión completo para la figura 9.28.

(*D*, *E*) (*F*). Otra posibilidad es usar (*A*, *C*) (*B*, *F*) (*D*, *E*) (*G*, *H*). No continuaremos con estas posibilidades y estudiaremos los aspectos relacionados con la especificación de las salidas tipo Mealy en el ejemplo 9.9.

Procedimiento de reducción de estados

Podemos resumir los pasos necesarios para generar la tabla de flujo reducida a partir de una tabla de flujo primitiva como sigue:

1. Usar el procedimiento de particionamiento para eliminar los estados equivalentes en una tabla de flujo primitiva.
2. Construir un diagrama de fusión para la tabla de flujo resultante.
3. Elegir subconjuntos de estados compatibles que puedan fusionarse, tratando de reducir al mínimo el número de subconjuntos necesario para cubrir todos los estados. Cada estado debe incluirse sólo en uno de los subconjuntos elegidos.
4. Derivar la tabla de flujo reducida mediante la fusión de las filas en los subconjuntos elegidos.
5. Repetir los pasos 2 a 4 para ver si son posibles reducciones posteriores.

Elegir un subconjunto óptimo de estados compatibles para fusión tal vez sea una tarea ardua porque para las FSM grandes puede haber muchas posibilidades que deben investigarse. El método de ensayo y error es una manera razonable de encarar este problema.

Ejemplo 9.8 Considere la tabla de flujo inicial de la figura 9.32. Para aplicar el procedimiento de particionamiento identificamos los pares de estados (*A*, *G*), (*B*, *L*) y (*H*, *K*) como filas potencialmente equivalentes porque ambas filas de cada par tienen las mismas salidas y sus entradas no-importa están en la misma columna. Las filas restantes son distintas a este respecto. Por tanto, la primera partición es

$$P_1 = (AG)(BL)(C)(D)(E)(F)(HK)(J)$$

Ahora los sucesores de (*A*, *G*) son (*A*, *G*) para $w_2w_1 = 00$, (*F*, *B*) para 01 y (*C*, *J*) para 10. Como *F* y *B*, igual que *C* y *J*, no están en el mismo bloque, se deduce que *A* y *G* no son equivalentes. Los sucesores de (*B*, *L*) son (*A*, *A*), (*B*, *L*) y (*H*, *K*), respectivamente. Todos están en bloques individuales. Los sucesores de (*H*, *K*) son (*L*, *B*), (*E*, *E*) y (*H*, *K*), los cuales están contenidos en bloques individuales. En consecuencia, la segunda partición es

$$P_2 = (A)(G)(BL)(C)(D)(E)(F)(HK)(J)$$

La repetición de la prueba del sucesor muestra que los sucesores de (*B*, *L*) y (*H*, *K*) aún están en bloques individuales; por consiguiente

$$P_3 = P_2$$

Combinar las filas *B* y *L* bajo el nombre de *B* y las filas *H* y *K* bajo el nombre de *H* conduce a la tabla de flujo de la figura 9.33.

Un diagrama de fusión para esta tabla de flujo se da en la figura 9.34. Indica que las filas *B* y *H* deben fusionarse en una fila, que etiquetaremos como *B*. El diagrama de fusión también sugiere que las filas *D* y *E* deben fusionarse; llamaremos a la nueva fila *D*. Las filas restantes presentan más de una posibilidad para la fusión. Las filas *A* y *F* pueden fusionarse, pero en ese

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	K	0
G	(G)	B	J	–	0
H	–	L	E	(H)	1
J	G	–	(J)	–	0
K	–	B	E	(K)	1
L	A	(L)	–	K	1

Figura 9.32 Tabla de flujo para el ejemplo 9.8.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	H	0
G	(G)	B	J	–	0
H	–	B	E	(H)	1
J	G	–	(J)	–	0

Figura 9.33 Reducción obtenida al usar el procedimiento de particionamiento.

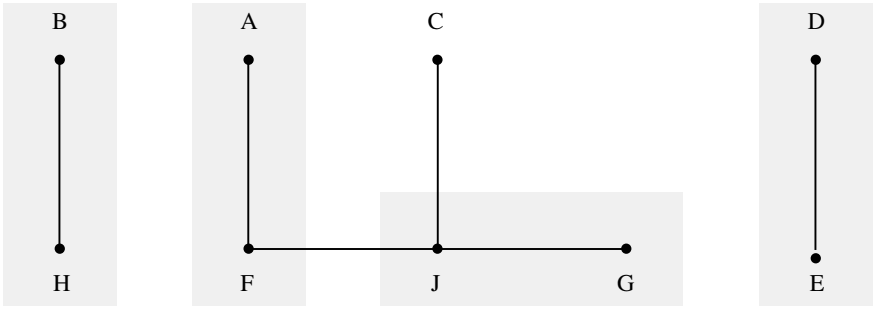


Figura 9.34 Diagrama de fusión para la figura 9.33.

caso *F* y *J* no pueden fusionarse. Las filas *C* y *J* pueden fusionarse, o *G* y *J* pueden hacerlo. Elegiremos fusionar las filas *A* y *F* en una fila nueva llamada *A* y las filas *G* y *J* en una fila nueva denominada *G*. La opción de fusión se indica en gris en el diagrama. La tabla de flujo resultante se muestra en la figura 9.35. Para ver si esta tabla ofrece otra oportunidad de fusión podemos construir el diagrama de fusión de la figura 9.36. A partir de éste es evidente que las filas *C* y *G* pueden fusionarse; sea la nueva fila *C*. Esto nos lleva a la tabla de flujo de la figura 9.37, que ya no puede reducirse.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	G	–	(C)	D	0
D	G	A	(D)	(D)	1
G	(G)	B	(G)	–	0

Figura 9.35 Reducción obtenida a partir del diagrama de fusión de la figura 9.34.

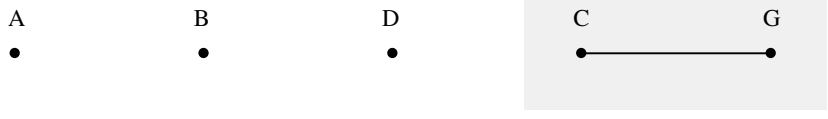


Figura 9.36 Diagrama de fusión para la figura 9.35.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	(C)	B	(C)	D	0
D	C	A	(D)	(D)	1

Figura 9.37 Tabla de flujo reducida para el ejemplo 9.8.

Ejemplo 9.9

Considere la tabla de flujo de la figura 9.38. Al aplicar el procedimiento de particionamiento a esta tabla obtenemos

$$P_1 = (AFK)(BJ)(CG)(D)(E)(H)$$
$$P_2 = (A)(FK)(BJ)(C)(G)(D)(E)(H)$$
$$P_3 = P_2$$

La combinación de B y J en un nuevo estado B, y F y K en F genera la tabla de flujo de la figura 9.39.

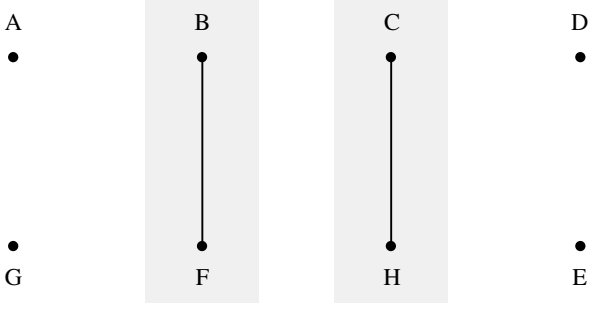
En la figura 9.40*a* se proporciona un diagrama de fusión para esta tabla de flujo y se indican las posibilidades para la fusión si se va a preservar el modelo Moore de la FSM. En este caso B y F pueden fusionarse, así como C y H, lo que resulta en una tabla de flujo de seis filas.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	G	E	–	0
B	K	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	J	–	0
G	K	(G)	–	D	1
H	–	–	E	(H)	1
J	F	–	(J)	D	0
K	(K)	C	B	–	0

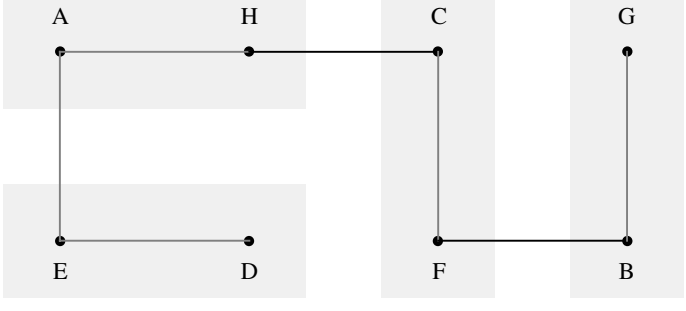
Figura 9.38 Tabla de flujo para el ejemplo 9.9.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	G	E	–	0
B	F	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	B	–	0
G	F	(G)	–	D	1
H	–	–	E	(H)	1

Figura 9.39 Reducción resultante del procedimiento de particionamiento.



a) Diagrama que preserva el modelo Moore



b) Diagrama de fusión completo

Figura 9.40 Diagramas de fusión para la figura 9.39.

Ahora debemos considerar las posibilidades de fusión si estamos dispuestos a cambiar al modelo Mealy. Cuando pasamos del modelo Moore al modelo Mealy, un estado estable en este último debe generar la misma salida que en el modelo Moore. También es importante asegurar que las transiciones en el modelo Mealy no producirán oscilaciones indeseables en la señal de salida.

En la figura 9.41 se indica cómo puede representarse la FSM de la figura 9.39 en la forma Mealy. Los valores de estado siguiente permanecen sin cambios. En la figura 9.41, para cada estado estable el valor de salida debe ser el mismo que para la fila correspondiente de la tabla tipo Moore. Por ejemplo, $z = 0$ cuando el estado A es estable bajo $w_2w_1 = 00$. Además, $z = 0$ cuando los estados B, D y F son estables bajo $w_2w_1 = 10$, 11 y 00, respectivamente. De manera similar, $z = 1$ cuando C, E, G y H son estables bajo $w_2w_1 = 01$, 10, 01 y 11, respectivamente. Si una transición de un estado estable a otro requiere que la salida cambie de 0 a 1, o de 1 a 0, entonces el momento exacto cuando ocurre el cambio no es importante, como explicamos en la sección 9.1 al estudiar la figura 9.3. Suponga que la FSM está estable en A bajo $w_2w_1 = 00$, lo que produce $z = 0$. Si las entradas luego cambian a $w_2w_1 = 01$, una transición al estado G debe hacerse, donde $z = 1$. Como no es esencial que z se vuelva 1 antes que el circuito alcance el estado G, el valor de salida en la fila A que corresponde a esta transición puede tratarse como una condición no-importa; por consiguiente, se deja sin especificar en la tabla. A partir del estado estable A también es posible cambiar a E, lo cual permite especificar otra condición no-importa, ya que z cambia de 0 a 1. Una situación diferente se presenta en la fila B. Digamos que el circuito se halla estable en B bajo $w_2w_1 = 10$ y que las entradas cambian a 11. Esto tiene que ocasionar un cambio al estado estable D y z debe permanecer en 0 a lo largo de todo el cambio en los estados. En consecuencia, la salida en la fila B bajo $w_2w_1 = 11$ se especifica como 0. Si se dejara sin especificar, para usarse como una condición no-importa, entonces es posible que en la implementación del circuito esta condición no-importa se trate como un 1, lo cual ocasionaría una oscilación en z, que cambiaría de $0 \rightarrow 1 \rightarrow 0$ a medida que el circuito pasa de B a D cuando las entradas cambian de 10 a 11. La misma situación ocurre para la transición de B a F cuando las entradas cambian

Estado presente	Estado siguiente				Salida z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	G	E	–	0	–	–	–
B	F	–	(B)	D	0	–	0	0
C	F	(C)	–	H	–	1	–	1
D	–	C	E	(D)	–	–	–	0
E	A	–	(E)	D	–	–	1	–
F	(F)	C	B	–	0	–	0	–
G	F	(G)	–	D	–	1	–	–
H	–	–	E	(H)	–	–	1	1

Figura 9.41 La FSM de la figura 9.39 especificada en la forma del modelo Mealy.

Estado presente	Estado siguiente				Salida z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	B	D	(A)	0	–	1	1
B	C	(B)	(B)	D	0	1	0	0
C	(C)	(C)	B	A	0	1	0	1
D	A	C	(D)	(D)	–	–	1	0

Figura 9.42 Tabla de flujo reducida para el ejemplo 9.9.

de 10 a 00. Podemos seguir el mismo razonamiento para determinar otros valores de salida en la figura 9.41.

A partir de la figura 9.41 podemos derivar el diagrama de fusión de la figura 9.40*b*. Las líneas grises conectan las filas que pueden fusionarse sólo si se especifica la salida en el estilo Mealy. Las líneas negras conectan las filas que pueden fusionarse incluso si las salidas son del tipo Moore; corresponden al diagrama de la figura 9.40*a*. Al elegir los subconjuntos de estados compatibles (*A*, *H*), (*B*, *G*), (*C*, *F*) y (*D*, *E*), la FSM puede representarse usando sólo cuatro estados. Al fusionar los estados *A* y *H* en un estado nuevo *A*, los estados *B* y *G* en *B*, los estados *C* y *F* en *C*, y *D* y *E* en *D* obtenemos la tabla de flujo reducida de la figura 9.42. Cada valor de esta tabla cumple los requisitos especificados en las filas correspondientes que fusionamos.

Ejemplo 9.10 Como otro ejemplo considere la tabla de flujo de la figura 9.43. El procedimiento de partición produce

$$P_1 = (AF)(BEG)(C)(D)(H)$$
$$P_2 = (AF)(BE)(G)(C)(D)(H)$$
$$P_3 = P_2$$

Estado presente	Estado siguiente				Salida z
	$w_2w_1 = 00$	01	10	11	
A	(A)	B	C	–	0
B	F	(B)	–	H	0
C	F	–	(C)	H	0
D	(D)	G	C	–	1
E	A	(E)	–	H	0
F	(F)	E	C	–	0
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figura 9.43 Tabla de flujo para el ejemplo 9.10.

Si se reemplaza el estado F con A y el estado E con B se obtiene la tabla de flujo de la figura 9.44. El diagrama de fusión correspondiente se presenta en la figura 9.45. Es evidente que los estados A, B y C pueden fusionarse y sustituirse con un estado A nuevo. Además, D, G y H pueden fusionarse en un estado nuevo D. El resultado es la tabla de flujo reducida de la figura 9.46, la cual tiene sólo dos filas. De nuevo hemos utilizado el modelo Mealy porque los estados estables fusionados D y H tienen $z = 1$ mientras que G tiene $z = 0$.

Estado presente	Estado siguiente				Salida z
	$w_2w_1 = 00$	01	10	11	
A	(A)	B	C	–	0
B	A	(B)	–	H	0
C	A	–	(C)	H	0
D	(D)	G	C	–	1
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figura 9.44 Reducción después del procedimiento de particionamiento.

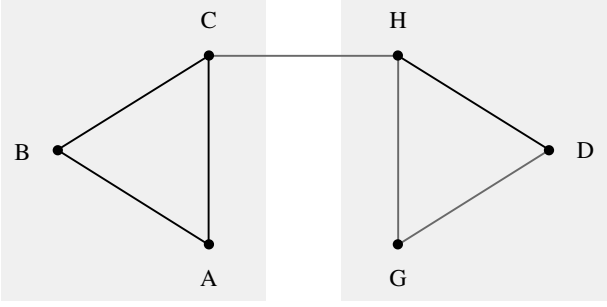


Figura 9.45 Diagrama de fusión para la figura 9.44.

Estado presente	Estado siguiente				Salida z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	(A)	(A)	D	0	0	0	–
D	(D)	(D)	A	(D)	1	0	–	1

Figura 9.46 Tabla de flujo reducida para el ejemplo 9.10.

9.5 ASIGNACIÓN DE ESTADOS

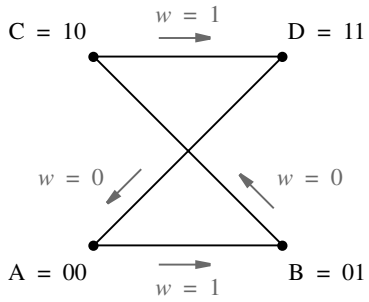
Los ejemplos de la sección 9.3 ilustran que la tarea de asignación de estado para las FSM asíncronas es compleja. El tiempo requerido para cambiar el valor de una variable de estado depende de los retrasos de propagación en el circuito. Por tanto, es imposible asegurar que un cambio en los valores de dos o más variables ocurrirá exactamente al mismo tiempo. Para lograr una operación confiable del circuito, las variables de estado deben cambiar sus valores una a la vez de manera controlada. Esto se logra diseñando el circuito de tal modo que un cambio de un estado a otro implique un cambio únicamente en una variable de estado.

Los estados de las FSM se codifican como cadenas de bits que representan diferentes combinaciones de las variables de estado. El número de posiciones de bits en las que difieren dos cadenas de bits dadas se llama la *distancia Hamming* entre las cadenas. Por ejemplo, para las cadenas de bits 0110 y 0100 la distancia Hamming es 1, mientras que para 0110 y 1101 es 3. Si seguimos esta terminología, una asignación de estados ideal tiene una distancia Hamming de 1 para todas las transiciones de un estado estable a otro. Cuando la asignación de estados ideal no es posible, debe buscarse una alternativa que emplee los estados sin especificar o las transiciones a través de estados inestables (o ambas cosas a la vez). A veces es necesario aumentar el número de variables de estado para proporcionar la flexibilidad necesaria.

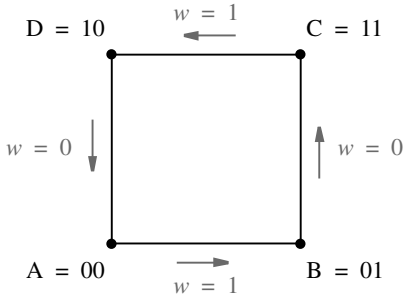
Ejemplo 9.11 Considere la FSM generadora de paridad de la figura 9.13. Dos posibles asignaciones de estados para esta FSM se presentan en la figura 9.14. Las transiciones entre estados, según se especifica en la figura 9.13*b*, pueden describirse gráficamente como se muestra en la figura 9.47. Cada fila de la tabla de flujo se representa con un punto. Los cuatro puntos requeridos para representar las filas se colocan como vértices de un cuadrado. Cada vértice tiene un código asociado que representa una combinación de las variables de estado, $y_2 y_1$. Los códigos mostrados en la figura, con $y_2 y_1 = 00$ en la esquina inferior izquierda y así sucesivamente, corresponden a las coordenadas del cubo bidimensional presentado en la sección 4.8. En la figura 9.47*a* se muestra lo que ocurre si se usa la asignación de estados de la figura 9.14*a*, esto es, si $A = 00$, $B = 01$, $C = 10$ y $D = 11$. Existe una transición de A a B si $w = 1$, la cual únicamente requiere un cambio en y_1. Una transición de C a D ocurre si $w = 1$, lo que también sólo precisa un cambio en y_1. Sin embargo, una transición de B a C ocasionada por $w = 0$ implica un cambio en los valores tanto de y_2 como de y_1. De forma similar, las dos variables de estado deben cambiar al pasar de D a A si $w = 0$. Un cambio en las dos variables corresponde a una ruta diagonal en el diagrama.

En la figura 9.47*b* se muestra el efecto de la asignación de estados de la figura 9.14*b*, la cual invierte las combinaciones asignadas a C y a D. En este caso las cuatro transiciones se hallan a lo largo de los bordes del cubo bidimensional y conllevan un cambio sólo en una de las variables de estado. Ésta es la asignación de estados deseada.

Ejemplo 9.12 En la figura 9.21*a* se observa la tabla de flujo para una FSM árbitro. Las transiciones para esta FSM se muestran en la figura 9.48*a*, donde se usa la asignación de estados $A = 00$, $B = 01$ y $C = 10$. En este caso son posibles múltiples transiciones entre los estados. Por ejemplo, hay dos transiciones entre A y B: de B a A si $r_2 r_1 = 00$ y de A a B si $r_2 r_1 = 01$. De nuevo hay una ruta diagonal, que corresponde a las transiciones entre B y C, las cuales deben evitarse. Una solución posible es introducir un cuarto estado, D, como se indica en la figura 9.48*b*. Ahora las transiciones entre B y C pueden ocurrir a través del estado inestable D. Por tanto, en



a) Correspondiente a la figura 9.14*a*



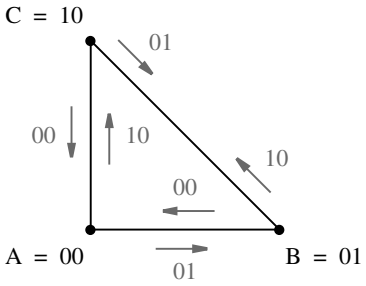
b) Correspondiente a la figura 9.14*b*

Figura 9.47 Transiciones de la figura 9.13.

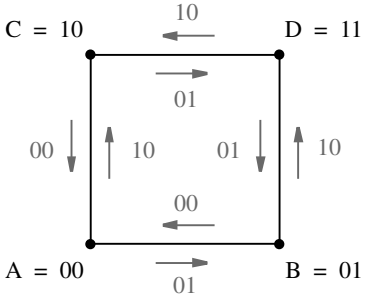
vez de ir directamente de *B* a *C* cuando $r_2r_1 = 10$, el circuito primero irá de *B* a *D* y luego de *D* a *C*.

Si se emplea el arreglo de la figura 9.48*b* es preciso modificar la tabla de flujo como se muestra en la figura 9.49. El estado *D* no es estable para ninguna combinación de entrada. No puede alcanzarse si $r_2r_1 = 00$ u 11; por consiguiente, estas combinaciones se dejan sin especificar en la tabla. También observe que hemos especificado la salida $g_2g_1 = 10$ para el estado *D*, en vez de dejarla sin especificar. Cuando ocurre una transición de un estado estable a otro a través de un estado inestable, la salida del estado inestable debe ser la misma que la salida de uno de los dos estados estables implicados en la transición para asegurar que no se genere una salida errónea al pasar por el estado inestable.

Es interesante comparar esta tabla de flujo con la tabla de excitación de la figura 9.21*b*, la cual también se basa en el uso del estado *D* adicional. En la figura 9.21*b* el estado *D* especifica las transiciones necesarias si el circuito se encuentra por accidente en este estado como resultado de una carrera cuando intentan cambiar los valores de las dos variables de estado. En la figura 9.49 el estado *D* se usa en transiciones ordenadas, que no son susceptibles a ninguna condición de carrera.



a) Transiciones en la figura 9.21*a*



b) Con el estado *D* adicional

Figura 9.48 Transiciones de la FSM árbitro de la figura 9.21.

Estado presente	Estado siguiente				Salida
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(A)	B	C	–	00
B	A	(B)	D	(B)	01
C	A	D	(C)	(C)	10
D	–	B	C	–	10

Figura 9.49 Tabla de flujo modificada con base en las transiciones de la figura 9.48*b*.

9.5.1 DIAGRAMA DE TRANSICIÓN

Un diagrama que ilustra las transiciones especificadas en una tabla de flujo se llama *diagrama de transición*, aunque en algunos libros se denomina *diagrama de estados adyacentes*. Estos diagramas ofrecen una ayuda práctica en la búsqueda de una asignación de estados adecuada.

Se logra una buena asignación de estados si el diagrama de transición no posee rutas diagonales. Una forma general de expresar este requisito consiste en decir que ha de ser posible *incrustar* el diagrama de transición en un cubo de k dimensiones, pues en un cubo todas las transiciones entre los vértices adyacentes suponen la distancia Hamming de 1. Idealmente, un diagrama de transición para una FSM con n variables de estado puede incrustarse en un cubo de n dimensiones, como sucede con los ejemplos de las figuras 9.47*b* y 9.48*b*. Si esto no es posible, entonces se vuelve necesario introducir variables de estado adicionales, como veremos en ejemplos posteriores.

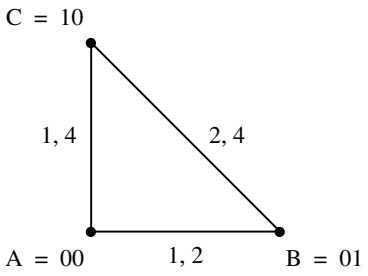
Los diagramas de las figuras 9.47 y 9.48 presentan toda la información pertinente para las transiciones entre los estados en las FSM dadas. Para FSM más grandes estos diagramas toman una apariencia abarrotada. En su lugar puede usarse una forma más simple, como describimos enseguida.

Un diagrama de transición debe mostrar las transiciones entre estados para cada combinación de las variables de entrada. La dirección de una transición, por ejemplo de A a B o de B a A, no es importante, ya que sólo es necesario asegurar que todas las transiciones suponen la distancia Hamming de 1. El diagrama de transición ha de mostrar el efecto de transiciones individuales en cada estado estable, lo cual puede implicar el paso a través de estados inestables. Para una fila específica de una tabla de flujo es posible tener dos o más entradas de estados estables para combinaciones de entrada diferentes. Es útil identificar las transiciones que llevan a esos estados estables con diferentes etiquetas en un diagrama de transición. Para dar a cada entrada de estado estable una etiqueta distinta, indicaremos las entradas de estado estable con los números 1, 2, 3,... Por tanto, si el estado A es estable para dos combinaciones de entrada, remplazaremos la etiqueta A con 1 para una combinación de entrada y con 2 para la otra.

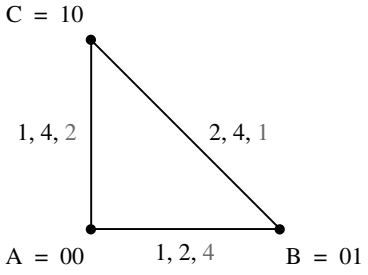
En la figura 9.50 se muestra una versión con etiquetas nuevas de la tabla de flujo de la figura 9.2l*a*. Hemos elegido arbitrariamente (A) como 1, las dos apariciones de (B) como 2 y 3, y las dos apariciones de (C) como 4 y 5. Todas las entradas de cada columna del estado siguiente se etiquetan siguiendo este esquema. Las transiciones identificadas por estas etiquetas se presentan en la figura 9.5l*a*. La misma información se proporciona en la figura 9.48*a*. En realidad, el diagrama de la figura 9.48*a* contiene más información porque las puntas de flecha muestran la dirección de cada transición. Obsérvese también que los bordes de ese diagrama se etiquetaron con los valores de entrada r_2r_1, mientras que a los bordes de la figura 9.51*a* se les pusieron etiquetas numéricas de estado estable como se explicó líneas arriba.

Estado presente	Estado siguiente				Salida
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(1)	2	4	–	00
B	1	(2)	4	(3)	01
C	1	2	(4)	(5)	10

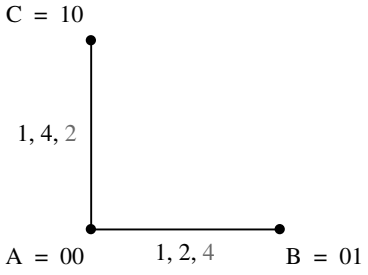
Figura 9.50 Tabla de flujo reetiquetada de la figura 9.21*a*.



a) Transiciones de la figura 9.50



b) Diagrama de transición completo



c) Diagrama de transición seleccionado

Figura 9.51 Diagramas de transición para la figura 9.50.

En la figura 9.50 se indica que el estado estable 2, que es una de las instancias del estado estable *B*, puede alcanzarse ya sea desde el estado a *A* o desde el estado *C*. Existe una etiqueta 2 correspondiente en las rutas que conectan los vértices del diagrama de la figura 9.51*a*. La dificultad desde el punto de vista de la asignación de estados es que la ruta de *C* a *B* es diagonal. En el ejemplo 9.12 este problema se resolvió introduciendo un nuevo estado *D*. Al examinar la tabla

de flujo de la figura 9.50 más de cerca se advierte que el comportamiento funcional de la FSM árbitro requerida puede lograrse si la transición de C a B ocurre a través del estado A. A saber, si el circuito está estable en C, entonces la entrada $r_2r_1 = 01$ puede provocar el cambio a A, desde el cual el circuito procede de inmediato al estado B. Podemos indicar la posibilidad de usar esta ruta colocando la etiqueta 2 en el borde que conecta C y A en la figura 9.51*a*.

Una situación parecida se observa para la transición de B a C, la cual se etiqueta como 4. Una ruta alternativa puede producirse si se hace que el circuito pase del estado B al A cuando $r_2r_1 = 10$ y luego de inmediato prosiga a C. Esto puede indicarse colocando la etiqueta 4 en el lado que conecta B y A en la figura 9.51*a*.

Siempre existe la posibilidad de tener otra ruta para una transición cuando dos estados tienen la misma etiqueta no circulada en el diagrama de flujo reetiquetado. En la figura 9.50 hay una tercera posibilidad como ésta si $r_2r_1 = 00$, usando la etiqueta 1. Esta posibilidad no es útil porque cambiar de B o C a A implica un cambio sólo en una variable de estado usando la asignación de estados de la figura 9.51*a*. Por consiguiente, no habrá beneficio en tener una transición entre B y C para esta combinación de entrada.

Para representar la posibilidad de tener otras rutas indicaremos en gris las transiciones correspondientes al diagrama. Por tanto, un diagrama de transición completo mostrará todas las transiciones directas a los estados estables en negro y las transiciones indirectas a través de los estados inestables en gris. En la figura 9.51*b* se muestra el diagrama de transición completo para la tabla de flujo de la figura 9.21*a*.

El diagrama de transición de la figura 9.51*b* no puede incrustarse en el cubo bidimensional porque algunas transiciones requieren una ruta diagonal. La etiqueta gris 1 en la ruta entre B y C no es de nuestro interés, pues sólo representa otra alternativa que no debe seguirse. Mas las transiciones entre B y C etiquetadas 2 y 4 sí se precisan. El diagrama muestra una ruta alternativa, a través de A, que tiene las etiquetas 2 y 4. Por ende, es posible usar la ruta alternativa y la conexión diagonal en el diagrama puede eliminarse. Esto conduce al diagrama de transición de la figura 9.51*c*, el cual puede incrustarse en el cubo bidimensional. La conclusión es que la asignación de estados $A = 00$, $B = 01$ y $C = 10$ es buena, pero la tabla de flujo debe modificarse para especificar las transiciones a través de las otras rutas. La tabla modificada es la misma que la tabla de flujo diseñada antes siguiendo un método distinto, y se muestra en la figura 9.23*a*.

Como comentario final en este ejemplo, observe el efecto que tienen las otras rutas en las salidas producidas por la FSM. Si $r_2r_1 = 01$, entonces un cambio del estado estable C a través del estado inestable A y luego al estado estable B genera las salidas $g_2g_1 = 10 \rightarrow 00 \rightarrow 01$, en vez de $10 \rightarrow 01$, según se especifica en la figura 9.2l*a*. Para la FSM árbitro esto no representa problema alguno, como se explicó en el ejemplo 9.6.

Procedimiento para derivar diagramas de transición

El diagrama de transición se deriva a partir de una tabla de flujo como sigue:

- Se deriva la tabla de flujo reetiquetada como se explicó antes. Para una combinación de entrada específica, todas las transiciones que conducen al mismo estado estable se etiquetan con el mismo número. Las transiciones a través de los estados inestables que finalmente conducen a un estado estable reciben el mismo número que la combinación del estado estable.
- Cada fila de la tabla de flujo se representa con un vértice.

- Se unen los dos vértices, V_i y V_j, por un borde si tienen el mismo número en cualquier columna de la tabla de flujo reetiquetada.
- Para cada columna en la que V_i y V_j tienen el mismo número, se etiqueta el borde entre V_i y V_j con ese número. Ocuparemos etiquetas negras para las transiciones directas a los estados (estables) encerrados en un círculo y etiquetas grises cuando las combinaciones del estado siguiente tanto para V_i como para V_j en la tabla de flujo no estén circuladas.

Obsérvese que el primer punto dice que en la tabla de flujo reetiquetada a las transiciones a través de estados inestables se les da la etiqueta del estado estable para el que conducen para una combinación de entrada dada. Por ejemplo, para derivar un diagrama de transición comenzando por la tabla de flujo de la figura 9.23*a*, la tabla se volvería a etiquetar para llegar a la tabla de la figura 9.50. La transición del estado estable *A* al estado estable *B*, cuando $r_2r_1 = 01$, tiene la etiqueta 2. La misma etiqueta se da para la transición del estado estable *C* al estado estable *A*, ya que esta transición a la larga conduce el estado estable *B*.

9.5.2 CÓMO APROVECHAR LAS COMBINACIONES DE ESTADO SIGUIENTE SIN ESPECIFICAR

Las entradas sin especificar en una tabla de flujo brindan un poco de flexibilidad en la búsqueda de asignaciones de estados adecuadas. En el ejemplo siguiente se presenta un posible enfoque. En él también se ilustran todos los pasos dados en la derivación de un diagrama de transición.

Ejemplo 9.13 Considere la tabla de flujo de la figura 9.52*a*. Esta FSM tiene siete entradas de estado estable. Si se les etiqueta en orden, de 1 a 7, se obtiene la tabla del inciso *b*) de la figura. En este caso los estados 1 y 2 corresponden al estado *A*, el 3 y el 4 al *B*, el 5 y 6 al *C* y el 7 al estado *D*. En la columna $w_2w_1 = 00$ hay una transición de *C* a *A*, que se etiqueta como 1, y una transición de *D* a *B*, que se etiqueta como 3, porque 1 y 3 son los estados estables sucesores en estas transiciones. De forma similar, en la columna 11 hay transiciones de *B* a *C* y de *D* a *A*, las cuales se etiquetan 6 y 2, respectivamente. En la columna 01 hay una transición de *A* a *B*, que se etiqueta 4. El estado *C* es estable para esta combinación de entrada; se etiqueta 5. No hay transición especificada que conduzca a este estado estable. El estado puede alcanzarse sólo si *C* es estable bajo $w_2w_1 = 11$, el cual se etiqueta 6, y luego las entradas cambian a $w_2w_1 = 01$. Advierta que la FSM permanece estable en *C* si las entradas cambian de 11 a 01, o viceversa. En la columna 10 se ilustra cómo se tratan los estados inestables. A partir del estado estable *A*, se especifica una transición al estado inestable *C*. En cuanto la FSM alcanza el estado *C* procede a cambiar al estado estable *D*, el cual se etiqueta 7. Por tanto, 7 se usa como la etiqueta para toda la secuencia de transición de *A* a *C* a *D*.

Al tomar las filas *A*, *B*, *C* y *D* como los cuatro vértices, un primer intento por trazar el diagrama de transición se presenta en la figura 9.53*a*. El diagrama muestra transiciones entre todos los pares de estados, lo que parece indicar que es imposible tener una asignación de estados donde todas las transiciones se caractericen por una distancia Hamming de 1. Si se usa la asignación de estados $A = 00$, $B = 01$, $C = 11$ y $D = 10$, entonces la transición diagonal entre *A* y *C*, o la de *B* y *D*, requiere que las dos variables de estado cambien sus valores. La ruta diagonal de *B* a *D* con la etiqueta 7 no es necesaria, ya que hay otra ruta de *B* a *D* con la etiqueta 7 que pasa

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(A)	B	C	(A)	00
B	(B)	(B)	D	C	01
C	A	(C)	D	(C)	10
D	B	–	(D)	A	11

a) Tabla de flujo

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(1)	4	7	(2)	00
B	(3)	(4)	7	6	01
C	1	(5)	7	(6)	10
D	3	–	(7)	2	11

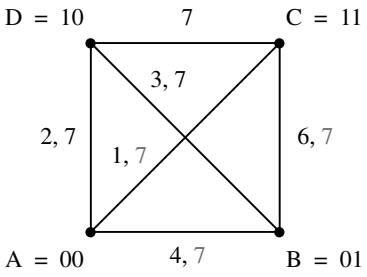
b) Tabla de flujo reetiquetada

Figura 9.52 Tablas de flujo para el ejemplo 9.13.

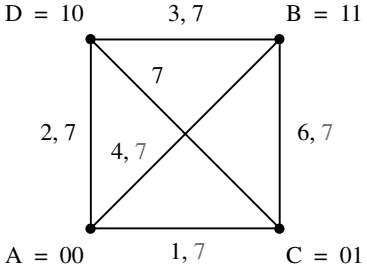
ya sea a través del estado *A* o del estado *C*. Lamentablemente, las rutas diagonales etiquetadas 1 y 3 no pueden eliminarse porque no hay rutas alternas para estas transiciones.

En nuestro siguiente intento por encontrar una asignación de estados adecuada, invertiremos los códigos dados a *B* y a *C*, lo que produce el diagrama de transición de la figura 9.53*b*. Ahora el mismo argumento sobre las rutas posibles etiquetadas como 7 indica que la diagonal de *C* a *D* puede omitirse. También pudiera omitirse la etiqueta 7 en la diagonal entre *A* y *B*. Sin embargo, esta diagonal debe conservarse a causa de la etiqueta 4 para la cual no hay ruta alterna entre *A* y *B*. Al estudiar la tabla de flujo de la figura 9.52*b* se observa una entrada sin especificar en la columna $w_2w_1 = 01$. Esta entrada puede aprovecharse si se le sustituye con la etiqueta 4, caso en el que la gráfica de transición mostraría la etiqueta 4 en los bordes que conectan *A* y *D*, así como *B* y *D*. Por tanto, la diagonal entre *A* y *B* podría eliminarse, lo que resultaría en el diagrama de transición de la figura 9.53*c*. Este diagrama puede incrustarse en un cubo bidimensional, lo que significa que la asignación de estados $A = 00$, $B = 11$, $C = 01$ y $D = 10$ puede emplearse.

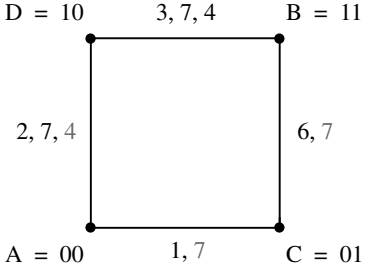
Para que el diagrama de transición de la figura 9.53*c* sea aplicable, la tabla de flujo para la FSM debe modificarse como se muestra en la figura 9.54*a*. La entrada sin especificar de la figura 9.52*a* ahora indica una transición al estado *B*. De acuerdo con la figura 9.53*c*, el cambio del estado *A* a *B* bajo la combinación de entrada $w_2w_1 = 01$ debe pasar a través del estado *D*; por ende,



a) Primer diagrama de transición



b) Segundo diagrama de transición



c) Diagrama de transición aumentado

Figura 9.53 Diagramas de transición para la figura 9.52.

la entrada correspondiente en la primera fila se modifica para asegurar que esto ocurra. Además, cuando $w_2w_1 = 10$, la FSM debe pasar al estado *D*. Si llega a estar en el estado *C*, entonces este cambio ha de ocurrir ya sea a través del estado *A* o del *B*. Hemos elegido la ruta a través del estado *B* en la figura 9.54*a*.

La tabla de flujo original de la figura 9.52*a* se define a partir del modelo Moore. La tabla de flujo modificada de la figura 9.54*a* requiere el uso del modelo Mealy porque las transiciones descritas antes a través de los estados inestables deben producir salidas correctas. Considere

Estado presente	Estado siguiente				Salida z_2z_1			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	D	D	(A)	00	00	11	00
B	(B)	(B)	D	C	01	01	11	01
C	A	(C)	B	(C)	−0	10	1−	10
D	B	B	(D)	A	−1	0−	11	00

a) Tabla de flujo modificada

	Estado presente y_2y_1	Estado siguiente				Salida			
		$w_2w_1 = 00$	01	10	11	00	01	10	11
		Y_2Y_1				z_2z_1			
A	00	(00)	10	10	(00)	00	00	11	00
B	11	(11)	(11)	10	01	01	01	11	01
C	01	00	(01)	11	(01)	−0	10	1−	10
D	10	11	11	(10)	00	−1	0−	11	00

b) Tabla de excitación

Figura 9.54 Realización de la FSM de la figura 9.52*a*.

primero el cambio desde A si $w_2w_1 = 01$. Mientras está estable en el estado A, el circuito debe producir la salida $z_2z_1 = 00$. Una vez que se alcanza el estado estable B, la salida debe volverse 01. El problema radica en que esta transición requiere una breve visita al estado D, lo que en el modelo Moore producirá $z_2z_1 = 11$; ello generaría una oscilación en la señal de salida z_2, la cual sufriría el cambio $0 \rightarrow 1 \rightarrow 0$. Para evitar este cambio no deseable, la salida en el estado D debe ser $z_2 = 0$ para esta combinación de entrada, la cual requiere que se utilice el modelo Mealy como se muestra en la figura 9.54*a*. Observe que mientras z_2 debe ser 0 en D para $w_2w_1 = 01$, z_1 puede ser 0 o 1, ya que está cambiando de 0 en el estado A a 1 en el estado B. Por consiguiente, z_1 puede dejarse sin especificar de modo que este caso pueda tratarse como una condición no-importa. Una situación similar surge cuando el circuito cambia de C a D a través de B si $w_2w_1 = 10$. La salida debe cambiar de 10 a 11, lo que significa que z_2 debe permanecer en 1 a lo largo de este cambio, incluido el breve periodo en el estado B donde la salida con el modelo Moore sería 01.

La tabla de flujo modificada y la asignación de estados elegida conduciría a la tabla de excitación de la figura 9.54*b*. A partir de esta tabla se derivan las expresiones de estado siguiente y de salida, como en los ejemplos de la sección 9.3.

9.5.3 ASIGNACIÓN DE ESTADOS USANDO VARIABLES DE ESTADO ADICIONALES

En la figura 9.52*a* hay una transición sin especificar que puede aprovecharse para encontrar una asignación de estados adecuada, como se mostró en la sección 9.5.2. En general, tal flexibilidad tal vez no exista. Puede ser imposible encontrar una asignación de estados sin carrera usando $\log_2 n$ variables de estado para una tabla de flujo que tiene n filas. El problema puede resolverse al añadir variables de estado adicionales, lo cual puede hacerse de tres formas, como se ilustra en los ejemplos siguientes.

Ejemplo 9.14 USO DE ESTADOS INESTABLES ADICIONALES Considere la FSM especificada por la tabla de flujo de la figura 9.55*a*, en cuyo inciso *b*) se muestra la misma tabla pero reetiquetada. El diagrama de transición correspondiente se representa en la figura 9.56*a*. En éste se indica que hay transiciones entre todos los pares de vértices (filas). Ningún reacomodo de los vértices existentes permitiría la asignación del diagrama de transición en un cubo de dos dimensiones.

Introduzcamos ahora una variable de estado más, de modo que podamos buscar una forma de asignar el diagrama de transición a un cubo de tres dimensiones. Con tres variables de estado la asignación para el estado A puede estar a una distancia Hamming de 1 diferente de las asignaciones para B, C y D. Por ejemplo, podríamos tener $A = 000$, $B = 001$, $C = 100$ y $D = 010$. Pero

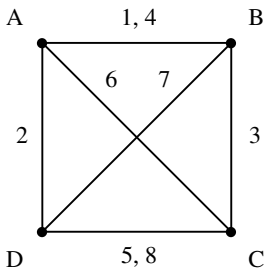
Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	D	(B)	01
C	(C)	B	(C)	D	10
D	C	A	(D)	(D)	11

a) Tabla de flujo

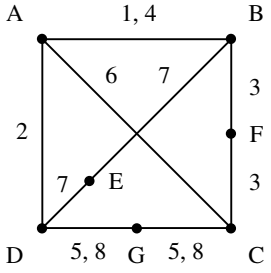
Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(1)	(2)	6	4	00
B	1	(3)	7	(4)	01
C	(5)	3	(6)	8	10
D	5	2	(7)	(8)	11

b) Tabla de flujo reetiquetada

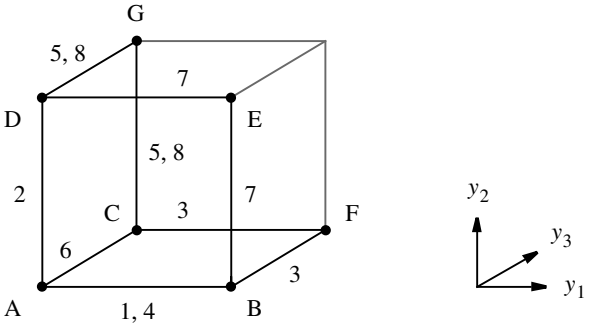
Figura 9.55 FSM para el ejemplo 9.14.



a) Diagrama de transición



b) Diagrama de transición aumentado



c) Diagrama de transición incrustado

Figura 9.56 Diagramas de transición para la figura 9.55.

entonces sería imposible tener los pares (*B*, *C*), (*B*, *D*) y (*C*, *D*) dentro de una distancia Hamming de 1. La solución aquí es insertar vértices adicionales en las rutas de transición, como se muestra en la figura 9.56*b*. El vértice *E* separa *B* de *D*, en tanto que los vértices *F* y *G* dividen las rutas (*B*, *C*) y (*C*, *D*). Las etiquetas asociadas con las transiciones se adjuntan en los dos segmentos de una ruta dividida. El diagrama de transición resultante puede incrustarse en un cubo tridimensional

como se indica en la figura 9.56*c*, donde la parte del cubo en negro comprende las rutas buscadas. Ahora la transición de *B* a *D* se lleva a cabo a través del vértice *E* si $w_2w_1 = 10$ (etiqueta 7). La transición de *C* a *B* ocurre a través de *F* si $w_2w_1 = 01$ (etiqueta 3). La transición de *C* a *D* pasa por *G* si $w_2w_1 = 11$ (etiqueta 8) y la transición de *D* a *C* pasa por *G* si $w_2w_1 = 00$ (etiqueta 5). Por tanto, la tabla de flujo ha de modificarse como se muestra en la figura 9.57*a*. Los tres estados adicionales son inestables porque el circuito no permanecerá en ellos para ninguna combinación

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	E	(B)	01
C	(C)	F	(C)	G	10
D	G	A	(D)	(D)	11
E	–	–	D	–	−1
F	–	B	–	–	01
G	C	–	–	D	1−

a) Tabla de flujo modificada

	Estado presente	Estado siguiente				Salida
		$w_2w_1 = 00$	01	10	11	
	$y_3y_2y_1$	$Y_3Y_2Y_1$				z_2z_1
A	000	(000)	(000)	100	001	00
B	001	000	(001)	011	(001)	01
C	100	(100)	101	(100)	110	10
D	010	110	000	(010)	(010)	11
E	011	–	–	010	–	−1
F	101	–	001	–	–	01
G	110	100	–	–	010	1−

b) Tabla de excitación

Figura 9.57 Tablas modificadas para el ejemplo 9.14.

de las entradas. El circuito simplemente pasará por estos estados en el proceso de cambiar de un estado estable a otro. Observe que cada uno de los estados E, F y G se necesita para facilitar las transiciones causadas por sólo una o dos combinaciones de las entradas. Por tanto, no es preciso especificar las acciones que podrían ser ocasionadas por otras combinaciones de entrada, ya que tales situaciones nunca ocurrirían en un circuito que funcione adecuadamente.

Las salidas de la figura 9.57*a* pueden especificarse con el modelo Mealy. Es esencial que se genere una salida apropiada cuando se pasa por los estados inestables a fin de evitar problemas técnicos indeseables en las señales de salida.

Si asignamos las variables de estado como se muestra a la derecha de la figura 9.56*c*, la tabla de flujo modificada conduce a la tabla de excitación de la figura 9.57*b*, a partir de la cual la derivación de las expresiones de estado siguiente y de salida es una tarea sencilla.

Ejemplo 9.15

USO DE PARES DE ESTADOS EQUIVALENTES Otro enfoque consiste en aumentar la flexibilidad en la asignación de estados al introducir un estado nuevo equivalente para cada estado existente. Por tanto, el estado A puede remplazarse con dos estados, $A1$ y $A2$, de tal manera que el circuito final produzca las mismas salidas para $A1$ y $A2$ que antes para A. De forma similar, los otros estados pueden remplazarse por pares de estado equivalentes. En la figura 9.58 se muestra cómo puede emplearse un cubo tridimensional a fin de hallar una buena asignación de estados para una tabla de flujo de cuatro filas. Los cuatro pares equivalentes se acomodan de forma que haya entre ellos la distancia Hamming mínima de 1. Por ejemplo, el par ($B1$, $B2$) tiene una distancia Hamming de 1 respecto a $A1$ (o $A2$), $C2$ y $D2$.

El diagrama de transición de la figura 9.56*a* puede incrustarse en el cubo tridimensional como se muestra en la figura 9.58. Puesto que existe una opción de dos vértices en el cubo por cada vértice en el diagrama de transición de la figura 9.56*a*, el diagrama de transición incrustado no supone ninguna ruta diagonal. Si se utiliza esta asignación de estados la tabla de flujo de la figura 9.55*a* debe modificarse como se presenta en la figura 9.59*a*. Las entradas de la tabla están hechas para permitir que cada transición en la tabla de flujo original se produzca usando una transición entre los pares correspondientes de estados equivalentes. Los dos estados en un par equivalente son estables para las combinaciones de entrada para las que el estado original es estable. Así, $A1$ y $A2$ son estables si $w_2w_1 = 00$ o 01, $B1$ y $B2$ son estables si $w_2w_1 = 01$ u 11 y así por el estilo. En cualquier instante la FSM puede estar en cualquiera de los dos estados equi-

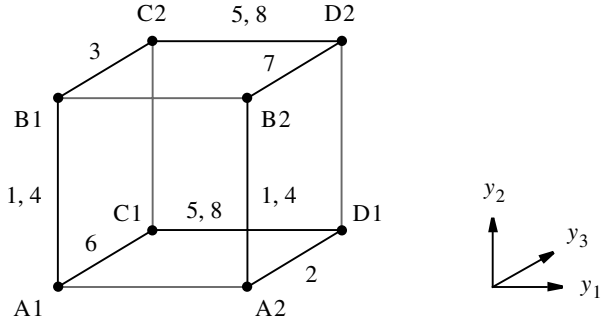


Figura 9.58 Diagrama de transición incrustado si se usan dos nodos por fila.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z_2z_1
A1	(A1)	(A1)	C1	B1	00
A2	(A2)	(A2)	A1	B2	00
B1	A1	(B1)	B2	(B1)	01
B2	A2	(B2)	D2	(B2)	01
C1	(C1)	C2	(C1)	D1	10
C2	(C2)	B1	(C2)	D2	11
D1	C1	A2	(D1)	(D1)	11
D2	C2	D1	(D2)	(D2)	11

a) Tabla de flujo modificada

	Estado presente	Estado siguiente				Salida
		$w_2w_1 = 00$	01	10	11	
	$y_3y_2y_1$	$Y_3Y_2Y_1$				z_2z_1
A1	000	(000)	(000)	100	010	00
A2	001	(001)	(001)	000	011	00
B1	010	000	(010)	011	(010)	01
B2	011	001	(011)	111	(011)	01
C1	100	(100)	110	(100)	101	10
C2	110	(110)	010	(110)	111	10
D1	101	100	001	(101)	(101)	11
D2	111	110	101	(111)	(111)	11

b) Tabla de excitación

Figura 9.59 Tablas de flujo y excitación modificadas para el ejemplo 9.15.

valentes que representan un estado original. Por tanto, un cambio de un estado a otro debe ser posible desde cualquiera de esos estados. Por citar un caso, en la figura 9.55*a* se especifica que la FSM debe cambiar del estado estable *A* al *B* si la entrada es $w_2w_1 = 11$. La transición equivalente en la tabla de flujo modificada es el cambio del estado *A*1 a *B*1 o de *A*2 a *B*2. Si la FSM se halla estable en *A* y la entrada cambia de 00 a 10, entonces se requiere un cambio a *C*. La transición equivalente en la tabla de flujo modificada es desde el estado *A*1 a *C*1; si ocurre que la FSM se encuentra en el estado *A*2, primero tendrá que cambiar a *A*1. Las entradas restantes de la figura 9.59*a* se derivan con el mismo razonamiento.

Las salidas se especifican mediante el modelo Moore, ya que los únicos estados estables son los involucrados en cambiar de un miembro del par equivalente a otro, y ambos miembros generan las mismas salidas. Por ejemplo, en la transición recién descrita de *A* a *C*, si el punto de partida es *A*2 es preciso ir primero a *A*1 y luego a *C*1. Aun cuando *A*1 es instable para $w_2w_1 = 10$, no hay problema porque su salida es la misma que la de *A*2. Por consiguiente, si la tabla de flujo original se define utilizando el modelo Moore, entonces la tabla modificada también puede hacerse con el mismo modelo.

El uso de la asignación de las variables de estado de la figura 9.58 proporciona la tabla de excitación de la figura 9.59*b*.

9.5.4 ASIGNACIÓN DE ESTADOS CON CODIFICACIÓN DE 1 ACTIVO

Los esquemas descritos líneas arriba basados en la incrustación de la tabla de flujo en un cubo pueden conducir a una asignación de estados óptima, pero requieren un enfoque de ensayo y error que se vuelve poco práctico para las máquinas grandes. Una alternativa directa, pero más costosa, es usar códigos de 1 activo. Si a cada fila de la tabla de flujo de una FSM se asigna un código de 1 activo, entonces pueden lograrse transiciones de estado sin carrera si se pasa por estados inestables que están a una distancia Hamming de 1 desde los dos estados estables involucrados en la transición. Supóngase que el código 0001 se asigna al estado *A* y el código 0010 al estado *B*. Por tanto, una transición libre de carrera de *A* a *B* puede pasar por un estado inestable 0011. De manera similar, si se asigna el código 0100 a *C*, entonces puede hacerse una transición de *A* a *C* a través del estado inestable 0101.

Al emplear este enfoque la tabla de flujo de la figura 9.55*a* puede modificarse como se ilustra en la figura 9.60. A los cuatro estados, *A*, *B*, *C* y *D*, se les asignan códigos de 1 activo. Como vemos en la figura, es necesario introducir seis estados inestables, de *E* a *J*, para manejar las transiciones necesarias. Estos estados inestables han de especificarse sólo para transiciones específicas, mientras que para otras combinaciones de entrada pueden tratarse como condiciones no-importa.

Las salidas pueden especificarse mediante el modelo Moore. En ciertos casos no tiene importancia cuando una señal de salida en particular cambia su valor. Por ejemplo, el estado *E* se usa para facilitar la transición del estado *A* al estado *C*. Como $z_2z_1 = 00$ en *A* y a 10 en *C*, no es relevante si z_2 cambia cuando pasa por el estado *E*.

Si bien es fácil de implementar, la codificación de 1 activo es cara porque requiere *n* variables de estado para implementar una tabla de flujo de *n* filas. ¡La simplicidad del diseño y el costo de la implementación a menudo proporcionan un compromiso desafiante en el diseño de los circuitos lógicos!

Asignación de estados	Estado presente	Estado siguiente				Salida
		$w_2w_1 = 00$	01	10	11	z_2z_1
0001	A	(A)	(A)	E	F	00
0010	B	F	(B)	G	(B)	01
0100	C	(C)	H	(C)	I	10
1000	D	I	J	(D)	(D)	11
0101	E	–	–	C	–	−0
0011	F	A	–	–	B	0−
1010	G	–	–	D	–	−1
0110	H	–	B	–	–	01
1100	I	C	–	–	D	1−
1001	J	–	A	–	–	00

Figura 9.60 Asignación de estados con codificación de 1 activo.

9.6 RIESGOS

En los circuitos secuenciales asíncronos es importante que no ocurran oscilaciones en las señales. El diseñador debe estar consciente de los posibles orígenes de éstas y cerciorarse de que las transiciones en un circuito estarán libres de errores. Los malfuncionamientos causados por la estructura de un circuito y los retrasos de propagación en éste se conocen como *riesgos*. En la figura 9.61 se ilustran dos tipos de riesgos.

Existe un *riesgo estático* si se supone que una señal permanece en un valor lógico específico cuando una variable de entrada cambia su valor, pero en vez de ello la señal sufre un cambio mo-

a) Riesgo estático

b) Riesgo dinámico

Figura 9.61 Definición de los riesgos.

mentáneo en su valor requerido. Como se muestra en la figura 9.61*a*, un tipo de riesgo estático ocurre cuando se supone que la señal en el nivel 1 permanecerá en 1 pero disminuye a 0 durante un breve lapso. Otro tipo es cuando se supone que la señal permanecerá en el nivel 0 pero aumenta momentáneamente a 1, con lo que se produce una oscilación.

Un tipo distinto de riesgo puede ocurrir cuando se supone que una señal cambiará de 1 a 0 o de 0 a 1. Si un cambio como éste implica una breve oscilación antes que la señal se establezca en su nivel nuevo, como se ilustra en la figura 9.61*b*, entonces se dice que existe un *riesgo dinámico*.

9.6.1 RIESGOS ESTÁTICOS

En la figura 9.62*a* se muestra un circuito con un riesgo estático. Supóngase que el circuito se halla en el estado donde $x_1 = x_2 = x_3 = 1$, caso en el que $f = 1$. Ahora suponga que x_1 cambia de 1 a 0. Entonces el circuito debiera mantener $f = 1$. Pero considérese lo que ocurre cuando los retrasos de propagación por las compuertas se tienen en cuenta. El cambio en x_1 probablemente se observará en el punto p antes que sea visto en el punto q porque la ruta de x_1 a q tiene una compuerta adicional (NOT). Por tanto, la señal en p se vuelve 0 antes que la señal en q se convierta en 1. Durante un breve lapso tanto p como q serán 0, lo que ocasionará que f caiga a 0 antes que se recupere de nuevo a 1. Esto da lugar a la señal descrita en el lado izquierdo de la figura 9.61*a*.

La oscilación en f puede impedirse como sigue. El circuito implementa la función

$$f = x_1x_2 + \bar{x}_1x_3$$

El mapa de Karnaugh correspondiente se observa en la figura 9.62*b*. Los dos términos producto producen los implicantes primos encerrados en negro. El riesgo recién explicado ocurre cuando hay una transición del implicante primo de x_1x_2 al implicante primo $\bar{x}_1x_3$. El riesgo puede eliminarse si se incluye el tercer implicante primo, encerrado en un círculo en gris. (Éste es el término de consenso, definido en la propiedad 17*a* de la sección 2.5.) Entonces la función se implementaría como

$$f = x_1x_2 + \bar{x}_1x_3 + x_2x_3$$

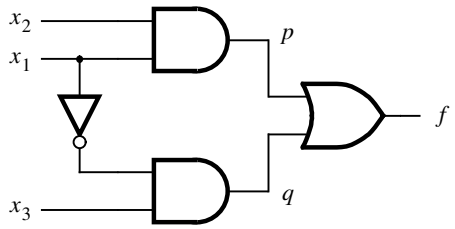
Ahora el cambio en x_1 de 1 a 0 no tendría efecto alguno en la salida porque el término producto x_2x_3 sería igual a 1 si $x_2 = x_3 = 1$, independientemente del valor de x_1. El circuito libre de riesgos resultante se representa en la figura 9.62*c*.

Un riesgo potencial existe siempre que dos 1 adyacentes en un mapa de Karnaugh no están cubiertos por un término producto simple. En consecuencia, una técnica para eliminar riesgos es encontrar una cobertura en la que algún término producto incluya cada par de 1 adyacentes. Por tanto, puesto que un cambio en una variable de entrada ocasiona una transición entre dos 1 adyacentes, no puede ocurrir ningún mal funcionamiento debido a que los dos 1 se incluyen en un término producto.

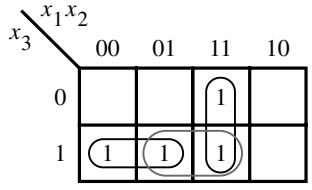
En los circuitos secuenciales asíncronos un riesgo puede hacer que el circuito cambie a un estado estable incorrecto. En el ejemplo 9.16 se ilustra esta situación.

Ejemplo 9.16

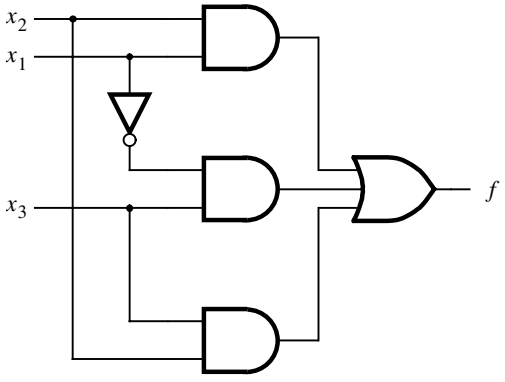
En el ejemplo 9.2 analizamos el circuito que funciona como un flip-flop D maestro-esclavo. A partir de la tabla de excitación de la figura 9.6*a* podría tratarse de sintetizar un circuito de costo mínimo que produzca las funciones requeridas, Y_m y Y_s. Esto daría



a) Circuito con un riesgo



b) Mapa de Karnaugh

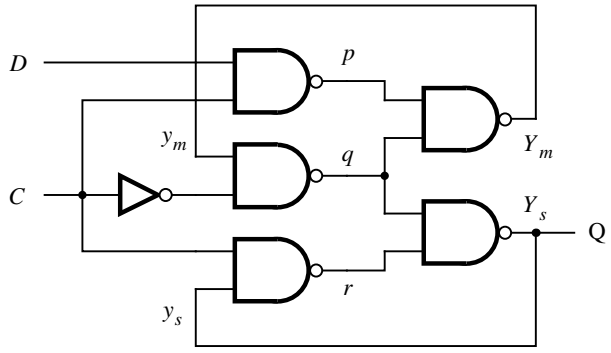


c) Circuito libre de riesgos

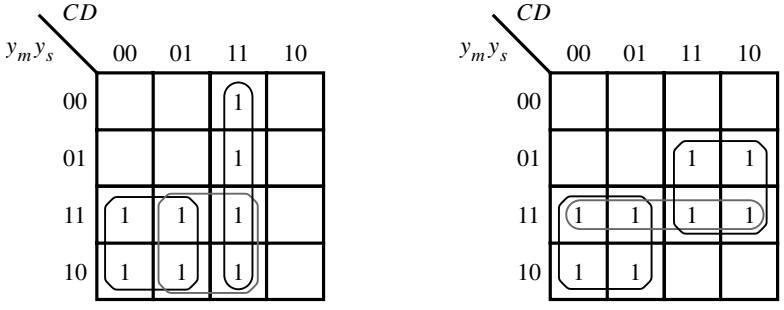
Figura 9.62 Un ejemplo de riesgo estático.

$$
\begin{aligned}
Y_m &= CD + \overline{C}y_m \\
&= (C \uparrow D) \uparrow (\overline{C} \uparrow y_m) \\
Y_s &= \overline{C}y_m + Cy_s \\
&= (\overline{C} \uparrow y_m) \uparrow (C \uparrow y_s)
\end{aligned}
$$

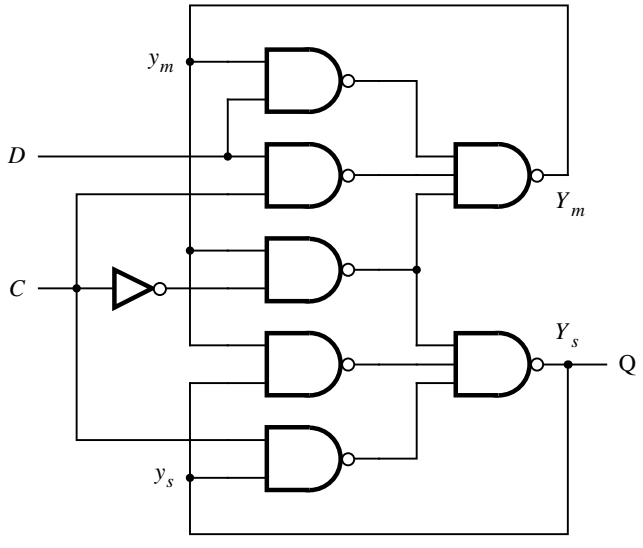
El circuito correspondiente se presenta en la figura 9.63*a*. A primera vista este circuito puede parecer más atractivo que los flip-flops estudiados en el capítulo 7, ya que es menos costoso. El problema es que contiene un riesgo estático.



a) Circuito de costo mínimo



b) Mapas de Karnaugh para Y_m y Y_s de la figura 9.6*a*



c) Circuito libre de riesgos

Figura 9.63 Implementación en dos niveles del flip-flop D maestro-esclavo.

En la figura 9.63*b* se muestran los mapas de Karnaugh para las funciones Y_m y Y_s. La implementación de costo mínimo se basa en los implicantes primos encerrados en negro. Para ver cómo afectan los riesgos estáticos este circuito, suponga que actualmente $Y_s = 1$ y $C = D = 1$. El circuito genera $Y_m = 1$. Ahora suponga que C cambia de 1 a 0. Para que el flip-flop se comporte adecuadamente, Y_s debe permanecer igual a 1. En la figura 9.63*a*, cuando C cambia a 0, tanto p como r se vuelven 1. Por el retraso a través de la compuerta NOT, q aún puede ser 1, lo que hace que el circuito genere $Y_m = Y_s = 0$. La retroalimentación desde Y_m conservará $q = 1$. Por consiguiente, el circuito permanece en un estado estable incorrecto con $Y_s = 0$.

Para evitar los riesgos también es necesario incluir los términos encerrados en gris, lo que da lugar a las expresiones

$$Y_m = CD + \overline{C}y_m + Dy_m$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

El circuito resultante, implementado con compuertas NAND, se muestra en la figura 9.63*c*.

Note que podemos obtener otra implementación con compuertas NAND si reescribimos las expresiones para Y_m y Y_s como

$$\begin{aligned} Y_m &= CD + (\overline{C} + D)y_m \\ &= (C \uparrow D) \uparrow ((\overline{C} + D) \uparrow y_m) \\ &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y_m) \\ Y_s &= \overline{C}y_m + (C + y_m)y_s \\ &= (\overline{C} \uparrow y_m) \uparrow ((\overline{C} \uparrow \overline{y}_m) \uparrow y_s) \end{aligned}$$

Estas expresiones corresponden exactamente al circuito de la figura 7.13.

Ejemplo 9.17 A partir de los ejemplos anteriores, parece que los riesgos estáticos pueden evitarse si se incluyen todos los implicantes primos en un circuito de suma de productos que realiza una función específica. Esto es cierto, pero no siempre es necesario incluir todos los implicantes primos, sino sólo los términos producto que cubren los pares de 1 adyacentes. No hay necesidad de cubrir lugares no-importa.

Considere la función de la figura 9.64. Un circuito libre de riesgos que implementa esta función debiera incluir los términos encerrados, lo cual da

$$f = \overline{x}_1 x_3 + x_2 x_3 + x_3 \overline{x}_4$$

El implicante primo $\overline{x}_1 \overline{x}_2$ no se precisa para evitar riesgos, ya que sólo se necesita para los dos 1 en la columna del extremo izquierdo. Estos 1 ya están cubiertos por $\overline{x}_1 x_3$.

Ejemplo 9.18 Los riesgos estáticos también pueden ocurrir en otros tipos de circuitos. En la figura 9.65*a* se describe un circuito de producto de sumas que contiene un riesgo. Si $x_1 = x_3 = 0$ y x_2 cambia de 0 a 1, entonces f debe permanecer en 0. Sin embargo, si la señal en p cambia antes que la señal en q, entonces p y q serán iguales a 1 por un breve lapso, causando la oscilación $0 \rightarrow 1 \rightarrow 0$ en f.

En un circuito producto de sumas, son las transiciones entre los 0 adyacentes lo que puede conducir a riesgos. Por tanto, para diseñar un circuito libre de riesgos, es preciso incluir términos

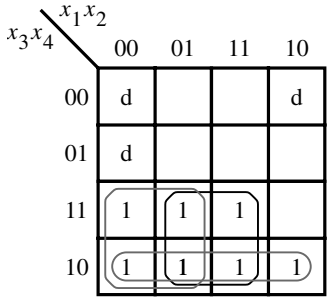


Figura 9.64 Función para el ejemplo 9.17.

suma que cubran todos los pares de 0 adyacentes. En este ejemplo el término en gris en el mapa de Kar-naugh debe incluirse, lo que resulta en

$$f = (x_1 + x_2)(\overline{x}_2 + x_3)(x_1 + x_3)$$

El circuito se muestra en la figura 9.65*c*.

9.6.2 RIESGOS DINÁMICOS

Un riesgo dinámico ocasiona malfuncionamientos en las transiciones $0 \rightarrow 1$ o $1 \rightarrow 0$ de una señal de salida. Un ejemplo se da en la figura 9.66. Si suponemos que todas las compuertas NAND tienen retrasos iguales, un diagrama de tiempo puede construirse como se muestra. El tiempo transcurrido entre dos líneas verticales corresponde a un retraso de compuerta. La salida *f* exhibe una oscilación que debe evitarse.

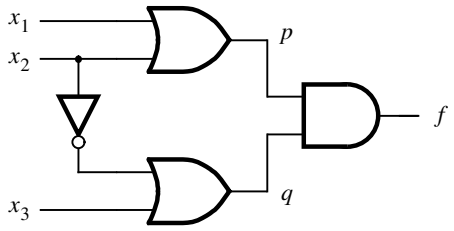
Resulta interesante considerar la función implementada por este circuito, la cual es

$$f = x_1\overline{x}_2 + \overline{x}_3x_4 + x_1x_4$$

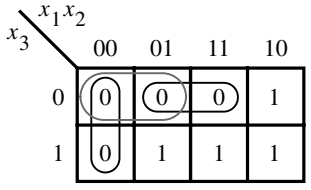
Ésta es la expresión de suma de productos de costo mínimo para la función. Si se implementara de esta forma, el circuito no tendría un riesgo estático ni uno dinámico.

Un riesgo dinámico es ocasionado por la estructura del circuito, donde existen múltiples rutas para que una señal dada cambie para propagarse por todas partes. Si la señal de salida cambia su valor tres veces, $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ en el ejemplo, entonces debe haber cuando menos tres rutas a lo largo de las cuales pueda propagarse un cambio desde una entrada primaria. Un circuito que tiene un riesgo dinámico también debe tener uno estático en alguna parte. Como se observa en la figura 9.66*b*, hay un riesgo estático que involucra la señal en el cable *b*.

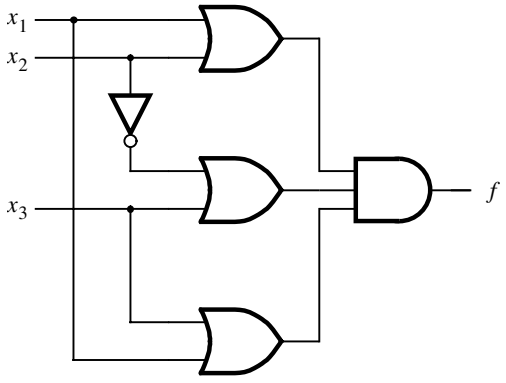
Los riesgos dinámicos se hallan en circuitos de múltiples niveles obtenidos usando las técnicas de factorización o descomposición estudiadas en el capítulo 4. Estos riesgos no son fáciles de detectar ni es sencillo lidiar con ellos. El diseñador puede evitar los riesgos dinámicos simplemente usando circuitos de dos niveles y cerciorándose de que no existan riesgos estáticos.



a) Circuito con un riesgo



b) Mapa de Karnaugh

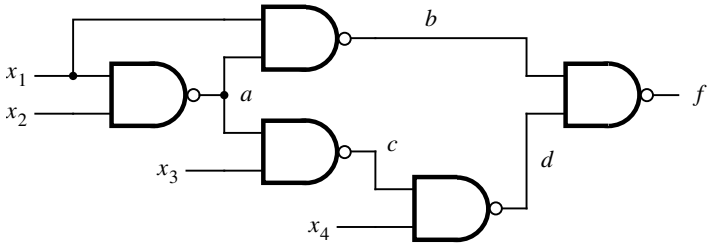


c) Circuito libre de riesgos

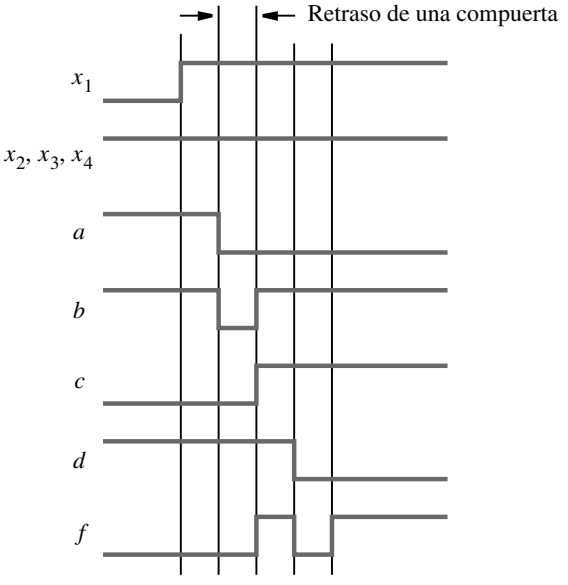
Figura 9.65 Riesgo estático en un circuito POS.

9.6.3 RELEVANCIA DE LOS RIESGOS

Un malfuncionamiento en un circuito secuencial asíncrono puede ocasionar que el circuito entre en un estado incorrecto y posiblemente se vuelva estable en ese estado. Por consiguiente, el sistema de circuitos que genera las variables de estado siguiente debe estar exento de riesgos. Basta eliminar los riesgos debidos a cambios en el valor de una sola variable porque la premisa básica



a) Circuito



b) Diagrama de tiempo

Figura 9.66 Circuito con un riesgo dinámico.

de un circuito secuencial asíncrono es que los valores tanto de las entradas primarias como de las variables de estado deben cambiar uno a la vez.

En los circuitos combinacionales, estudiados en los capítulos 4 a 6, no nos preocupamos por los riesgos, ya que la salida de un circuito depende únicamente de los valores de las entradas. En los circuitos secuenciales síncronos las señales de entrada deben ser estables dentro de los tiempos de preparación y espera de los flip-flops. No importa si los problemas ocurren fuera de tales tiempos respecto a la señal de reloj.

9.7 UN EJEMPLO DE DISEÑO COMPLETO

En las secciones anteriores examinamos varios aspectos de diseño de los circuitos secuenciales asíncronos. En esta sección daremos un ejemplo de diseño completo, que abarca todos los pasos necesarios.

9.7.1 EL CONTROLADOR DE LA MÁQUINA EXPENDEDORA

El mecanismo de control de una máquina expendedora es un buen vehículo para ilustrar una aplicación posible de un circuito digital. Lo utilizamos en el entorno síncrono en el capítulo 8. Un pequeño ejemplo de una máquina expendedora sirvió como objeto de análisis en la sección 9.2. Ahora consideraremos un controlador de máquina expendedora parecido al del ejemplo 8.6 para ver cómo podemos implementarlo mediante un circuito secuencial asíncrono. La especificación para el controlador es la siguiente:

- Acepta monedas de 5 y 10 centavos.
- Se precisa un total de 15 centavos para que la máquina suelte el caramelo.
- No se da cambio si se depositan 20 centavos.

Las monedas se depositan una por una. El mecanismo detector de monedas genera las señales $N = 1$ y $D = 1$ cuando ve una moneda de 5 centavos o una de 10, respectivamente. Es imposible tener $N = D = 1$ al mismo tiempo. Después de insertar una moneda para la cual la suma es igual o mayor que 15 centavos, la máquina suelta el caramelo y regresa al estado inicial.

En la figura 9.67 se muestra un diagrama de estado para la FSM requerida. Se derivó siguiendo un enfoque sencillo en el que todas las secuencias posibles de depositar monedas de 5 y 10 centavos se enumeran en una estructura tipo árbol. Para no abarrotar el diagrama, las etiquetas D y N indican las condiciones de entrada $DN = 10$ y $DN = 01$, respectivamente. La condición $DN = 00$ se etiquetó simplemente como 0. El caramelo se suelta en los estados F, H y K, los cuales se alcanzan después que se depositan 15 centavos, y en los estados I y L, una vez que se hace un depósito de 20 centavos.

La tabla de flujo correspondiente se proporciona en la figura 9.68. Puede reducirse usando el procedimiento de particionamiento como sigue

$$P_1 = (ADGJ)(BE)(C)(FIL)(HK)$$
$$P_2 = (A)(D)(GJ)(B)(E)(C)(FIL)(HK)$$
$$P_3 = P_2$$

Al usar G para representar los estados equivalentes G y J, F para representar F, I y L, y H para representar H y K, se genera la tabla de flujo parcialmente reducida de la figura 9.69. El diagrama de fusión para esta tabla se presenta en la figura 9.70. Indica que los estados C y E pueden fusionarse, igual que F y H. Por ende, la tabla de flujo reducida se obtiene como se muestra en la figura 9.71*a*. La misma información se representa en la forma de un diagrama de estado en la figura 9.72.

A continuación debe hallarse una asignación de estados adecuada. La tabla de flujo se reetiquetó en la figura 9.71*b* para asociar un número único a cada estado estable. Enseguida se obtiene el diagrama de transición de la figura 9.73*a*. Como deseamos tratar de incrustar el diagrama en un cubo tridimensional, se muestran ocho vértices en la figura. El diagrama muestra dos tran-

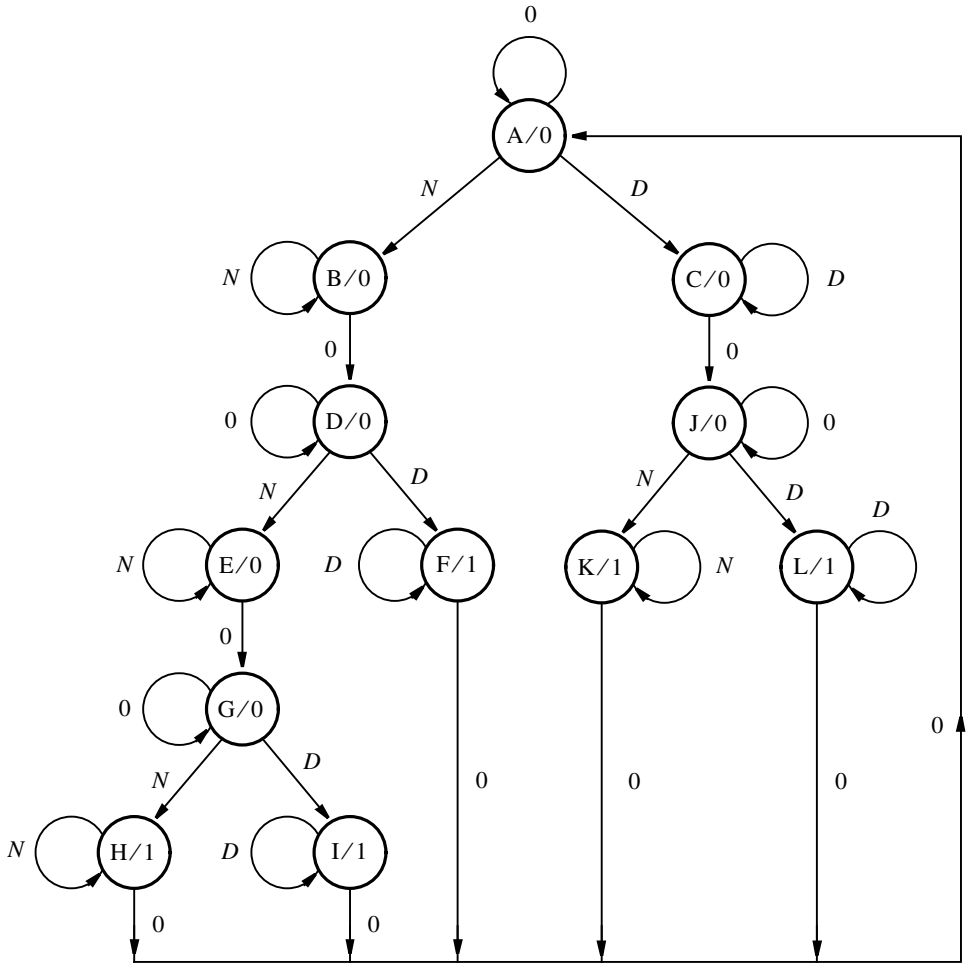


Figura 9.67 Diagrama de estado inicial para el controlador de la máquina expendedora.

siciones diagonales. La transición entre D y G (etiqueta 7) no-importa, pues sólo es una ruta alterna. La transición de A a C (etiqueta 4) es imprescindible y puede producirse a través de estados sin usar como se indica con gris en la figura 9.73*b*. Por tanto, el diagrama de transición puede incrustarse en un cubo tridimensional como se muestra. La tabla de excitación de la figura 9.74 se deriva mediante la asignación de estados de la figura 9.73*b*.

Los mapas de Karnaugh para las funciones de estado siguiente se presentan en la figura 9.75. A partir de estos mapas se obtienen las expresiones libres de riesgos

$$Y_1 = \overline{N}y_2 + Ny_1 + Dy_1 + y_1y_3 + y_1y_2$$
$$Y_2 = N\overline{y}_1 + Ny_2 + \overline{y}_1y_3 + \overline{D}y_2\overline{y}_3 + Dy_2y_3$$
$$Y_3 = D\overline{y}_1 + y_2y_3 + Ny_1y_2 + \overline{D}y_3\overline{N}$$

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	J	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	I	–	0
H	A	(H)	–	–	1
I	A	–	(I)	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	–	(L)	–	1

Figura 9.68 Tabla de flujo inicial para el controlador de la máquina expendedora.

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	F	–	0
H	A	(H)	–	–	1

Figura 9.69 Primer paso en la minimización de estados.

Todos los términos producto en estas expresiones son necesarios para la implementación POS de costo mínimo excepto para $y_1 y_2$, el cual se incluye para evitar riesgos en la expresión para Y_1. La expresión de salida es

$$z = y_1 \bar{y}_2 \bar{y}_3$$

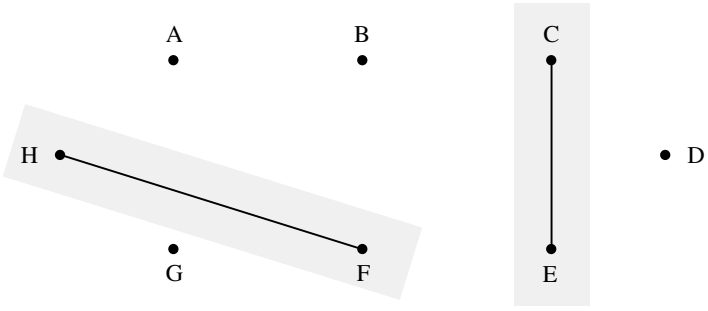


Figura 9.70 Diagrama de fusión para la figura 9.69.

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	(C)	(C)	–	0
D	(D)	C	F	–	0
F	A	(F)	(F)	–	1
G	(G)	F	F	–	0

a) Tabla de flujo minimizada

Estado presente	Estado siguiente				Salida
	$DN = 00$	01	10	11	z
A	(1)	2	4	–	0
B	5	(2)	–	–	0
C	8	(3)	(4)	–	0
D	(5)	3	7	–	0
F	1	(6)	(7)	–	1
G	(8)	6	7	–	0

b) Tabla de flujo reetiquetada

Figura 9.71 Tablas de flujo reducidas.

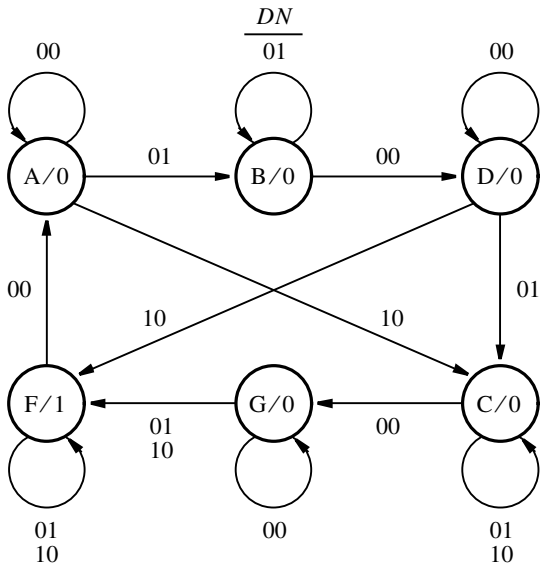
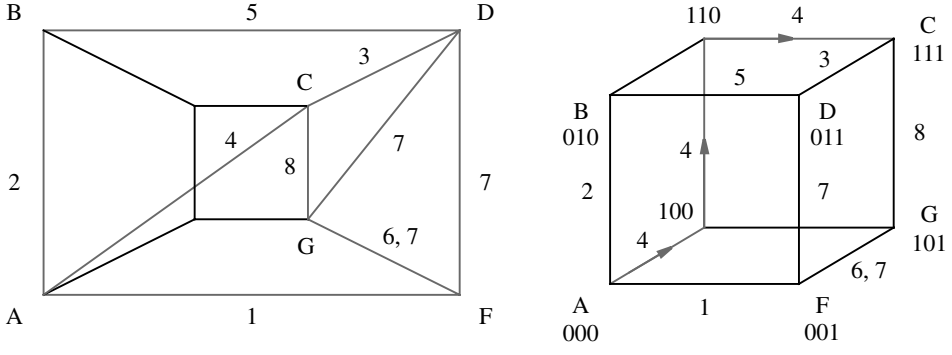


Figura 9.72 Diagrama de estado para el controlador de la máquina expendedora.



a) Diagrama de transición *b*) Incrustado en el cubo

Figura 9.73 Determinación de la asignación de estados.

	Estado presente $y_3y_2y_1$	Estado siguiente $Y_3Y_2Y_1$ $DN = 00$	01	10	11	Salida z
A	000	(000)	010	100	–	0
B	010	011	(010)		–	0
C	111	101	(111)	(111)	–	0
D	011	(011)	111	001	–	0
F	001	000	(001)	(001)	–	1
G	101	(101)	001	001	–	0
	100	–	–	110	–	0
	110	–	–	111	–	0

Figura 9.74 Tabla de excitación basada en la asignación de estados de la figura 9.73*b*.

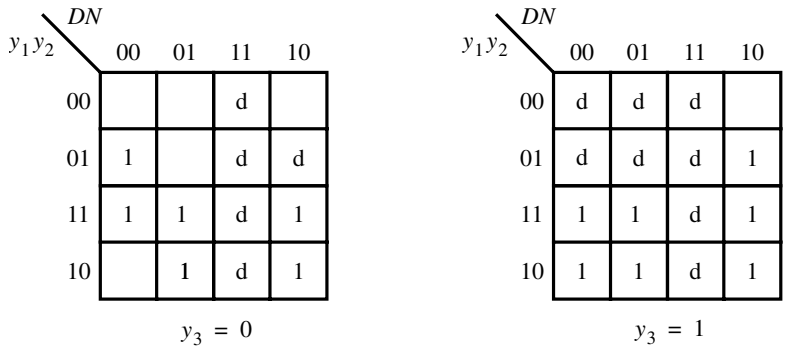
9.8 COMENTARIOS FINALES

Los circuitos secuenciales asíncronos son más difíciles de diseñar que los secuenciales síncronos. Las dificultades con las condiciones de carrera plantean un problema que debe manejarse con cuidado. En la actualidad hay poco apoyo de herramientas CAD para el diseño de circuitos secuenciales asíncronos. Por estas razones, la mayoría de los diseñadores recurre a los circuitos secuenciales síncronos en las aplicaciones prácticas.

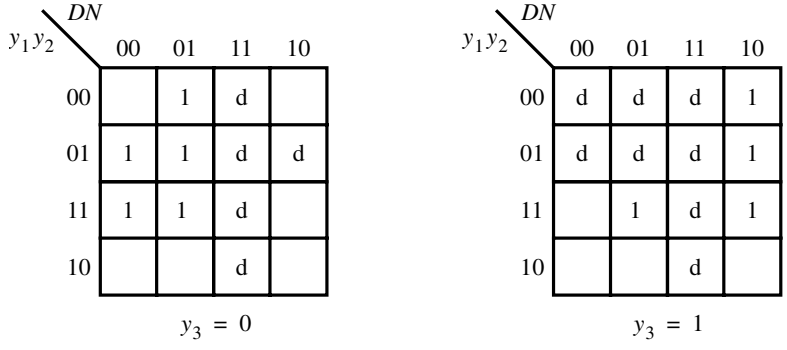
Una ventaja importante de los circuitos asíncronos es su velocidad de operación. Como no hay un reloj involucrado, la velocidad de operación depende únicamente de los retrasos de propagación en el circuito. En un sistema asíncrono que comprende varios circuitos, algunos de ellos funcionan más rápido que otros, lo que mejora potencialmente el desempeño general del sistema. Por el contrario, en los sistemas síncronos el periodo del reloj debe ser lo suficientemente largo para acomodarse al circuito más lento y esto tiene un efecto significativo en el desempeño.

Las técnicas de los circuitos asíncronos también son útiles en el diseño de sistemas que se componen de dos o más circuitos síncronos que operan bajo el control de diferentes relojes. Las señales que intercambian esos circuitos suelen parecer de naturaleza asíncrona.

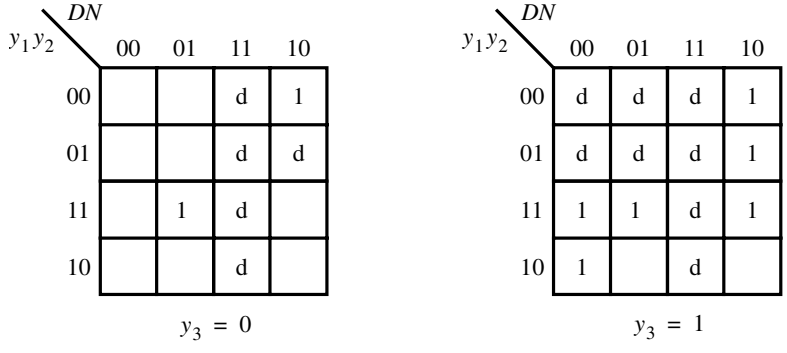
Desde el punto de vista del lector, es útil ver los circuitos asíncronos como un vehículo excelente para entender mejor el funcionamiento de los circuitos digitales en general. Estos circuitos ilustran las consecuencias de los retrasos de propagación y las condiciones de carrera que pueden ser inherentes a la estructura de un circuito. También ilustran el concepto de estabilidad, demostrado a lo largo de la existencia de estados estables e inestables. Para estudiar más acerca de los circuitos secuenciales asíncronos, el lector puede remitirse a las referencias [1-6].



a) Mapa para Y_1



b) Mapa para Y_2



c) Mapa para Y_3

Figura 9.75 Mapas de Karnaugh para las funciones de la figura 9.74.

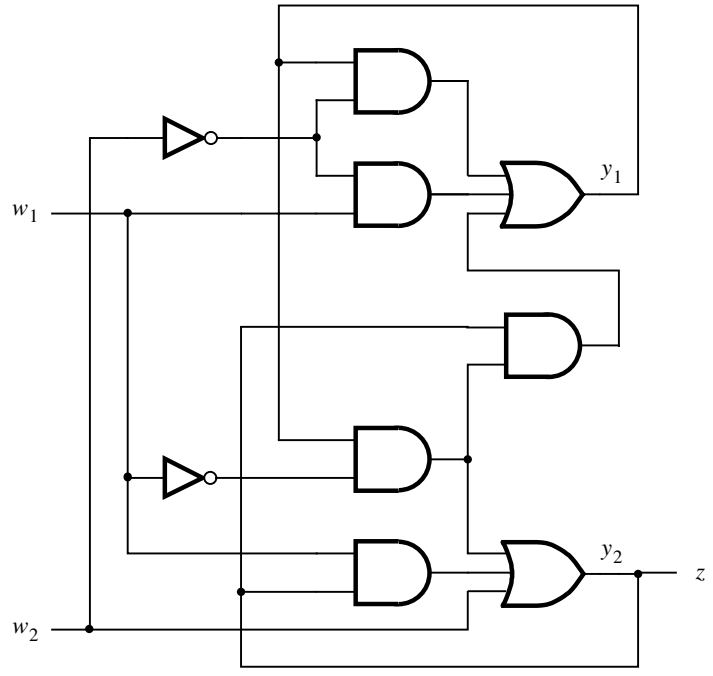


Figura 9.76 Circuito para el ejemplo 9.19.

9.9 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas comunes que el lector puede encontrar y se muestra cómo resolverlos.

Ejemplo 9.19

Problema: Derive una tabla de flujo que describa el comportamiento del circuito de la figura 9.76.

Solución: Al modelar el retraso de propagación de las compuertas del circuito como se muestra en la figura 9.8, el circuito de la figura 9.76 puede describirse por medio de las expresiones de estado siguiente y de salida que se indican a continuación

$$Y_1 = w_1\overline{w}_2 + \overline{w}_2 y_1 + \overline{w}_1 y_1 y_2$$
$$Y_2 = w_2 + \overline{w}_1 y_1 + w_1 y_2$$
$$z = y_2$$

Estas expresiones conducen a la tabla de excitación de la figura 9.77*a*. Si suponemos la asignación de estados $A = 00$, $B = 01$, $C = 10$ y $D = 11$ se produce la tabla de flujo de la figura 9.77*b*.

Estado presente y_2y_1	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	(00)	01	10	10	0
01	11	(01)	10	10	0
10	00	11	(10)	(10)	1
11	(11)	(11)	(11)	10	1

a) Tabla de excitación

Estado presente	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
A	(A)	B	C	C	0
B	D	(B)	C	C	0
C	A	D	(C)	(C)	1
D	(D)	(D)	(D)	C	1

b) Tabla de flujo implementada por el circuito

Estado presente	Estado siguiente $w_2w_1 = 00$	01	10	11	Salida z
A	(A)	B	C	C	0
B	D	(B)	–	C	0
C	A	D	(C)	(C)	1
D	(D)	(D)	(D)	C	1

c) Tabla de flujo final

Figura 9.77 Tablas de flujo y excitación para el circuito de la figura 9.76.

Como las entradas al circuito en un estado estable sólo pueden cambiar una por una, algunas entradas de la tabla de flujo pueden diseñarse sin especificar. Éste es el caso cuando el circuito se halla estable en el estado B y los valores de entrada son $w_2w_1 = 01$. Ahora, las dos entradas no pueden cambiar al mismo tiempo, lo que significa que la entrada correspondiente en la tabla de flujo puede diseñarse sin especificar. Sin embargo, una situación distinta se presenta cuando el circuito está estable en el estado A y los valores de entrada son $w_2w_1 = 00$. En este caso no podemos indicar la transición en la columna $w_2w_1 = 11$ como sin especificar. La razón es que si el circuito se encuentra estable en el estado B, debe poder cambiar al estado C cuando w_2 cambia de 0 a 1. Los estados B y C se implementan como $y_2\,y_1 = 01$ y $y_2\,y_1 = 10$, respectivamente. Puesto que ambas variables de estado deben cambiar sus valores, la ruta de 01 a 10 se llevará a cabo ya sea por 11 o por 00, según los retrasos en las diferentes trayectorias del circuito. Si y_2 cambia primero, el circuito pasará por el estado inestable D y luego se establecerá en el estado estable C. Pero si w_1 cambia primero, el circuito habrá de pasar por el estado inestable A antes de llegar al estado C. Por consiguiente, debe especificarse la transición al estado C en la primera fila. Éste es un ejemplo de una carrera segura, donde el circuito alcanza el estado destino correcto sin importar los retrasos de propagación de las diferentes trayectorias del circuito. La tabla de flujo final se presenta en la figura 9.77*c*.

Ejemplo 9.20

Problema: ¿Existe algún riesgo en el circuito de la figura 9.76?

Solución: En la figura 9.78 se proporcionan los mapas de Karnaugh para las expresiones de estado siguiente derivadas en el ejemplo 9.19. Como vemos en los mapas, todos los implicantes primos se incluyen en la expresión para Y_1. Pero la expresión para Y_2 incluye sólo tres de los cuatro implicantes primos disponibles. Hay un riesgo estático cuando $w_2y_2y_1 = 011$ y w_1 cambia de 0 a 1 (o de 1 a 0). Este riesgo puede eliminarse al añadir el cuarto implicante primo, $y_1\,y_2$ a la expresión para Y_2.

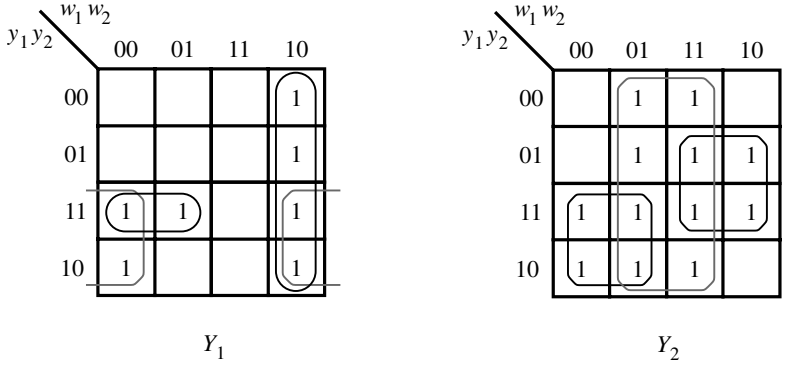


Figura 9.78 Mapas de Karnaugh para el circuito de la figura 9.76.

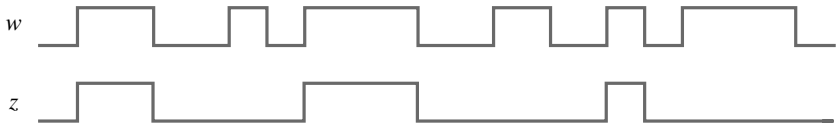
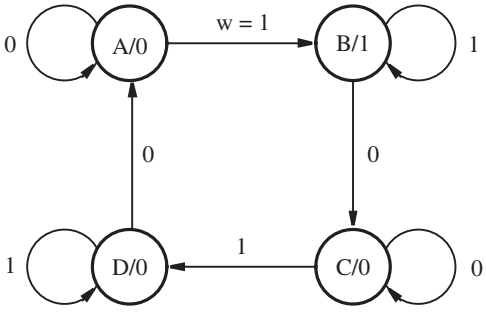


Figura 9.79 Formas de onda para el ejemplo 9.21.

Ejemplo 9.21 **Problema:** Un circuito tiene una entrada w y una entrada z. Una secuencia de pulsos se aplica en la entrada w. La salida ha de duplicarse cada segundo pulso, como se ilustra en la figura 9.79. Diseñe un circuito estable.

Solución: En la figura 9.80 se muestra un diagrama de estado posible y la tabla de flujo correspondiente. Compare esto con la FSM definida en el ejemplo 9.4 de la figura 9.13, que especifica un generador de paridad serial. La única diferencia es la señal de salida. En nuestro caso, $z = 1$ sólo en el estado B. Por consiguiente, las expresiones de estado siguiente son las mismas que las del ejemplo 9.4. La expresión de salida es

$$z = y_1\bar{y}_2$$



a) Diagrama de estado

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	0
D	A	(D)	0

b) Tabla de flujo

Figura 9.80 Diagrama de estado y tabla de flujo para el ejemplo 9.21.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	E	C	–	0
B	–	E	H	(B)	1
C	G	–	(C)	F	0
D	A	(D)	–	B	1
E	G	(E)	–	B	0
F	–	D	C	(F)	0
G	(G)	E	C	–	0
H	A	–	(H)	B	1

Figura 9.81 Tabla de flujo para el ejemplo 9.22.

Ejemplo 9.22

Problema: Considere la tabla de flujo de la figura 9.81. Reduzca esa tabla y encuentre una asignación de estados que permita realizar esta FSM de la manera más simple posible, conservando el modelo Moore. Derive una tabla de excitación.

Solución: Al usar el procedimiento de particionamiento en la tabla de flujo de la figura 9.81 se obtiene

$$P_1 = (ACEFG)(BDH)$$
$$P_2 = (AG)(B)(C)(D)(E)(F)(H)$$
$$P_3 = P_2$$

La combinación de A y G produce la tabla de flujo de la figura 9.82; en la 9.83 aparece un diagrama de fusión para esta tabla. La fusión de los estados (A, E), (C, F) y (D, H) conduce a la tabla de flujo reducida de la figura 9.84. Para encontrar una asignación de estados apropiada, reetiquetamos esta tabla de flujo como se indica en la figura 9.85 y construimos el diagrama de transición de la figura 9.86*a*. El único problema en este diagrama es la transición del estado D al A, etiquetado como 1. Un cambio de D a A puede hacerse a través del estado C si lo especificamos así en la tabla de flujo. Por tanto, no es necesaria una transición directa de D a A, como se describe en la figura 9.86*b*. La tabla de flujo resultante y la tabla de excitación correspondiente se muestran en la figura 9.87.

Ejemplo 9.23

Problema: Derive una implementación SOP de costo mínimo y libre de riesgos para la función

$$f(x_1, \ldots, x_5) = \sum m(2, 3, 14, 17, 19, 25, 26, 30) + D(10, 23, 27, 31)$$

Solución: El mapa de Karnaugh para la función se proporciona en la figura 9.88. A partir de éste, la expresión requerida se deriva como

$$f = x_1\bar{x}_3x_5 + x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_2\bar{x}_3x_4x_5$$

Los primeros tres términos producto cubren todos los 1 del mapa. El cuatro término es necesario para evitar un riesgo cuando $x_2x_3x_4x_5 = 0011$ y x_1 cambia de 0 a 1 (o de 1 a 0). Por tanto, cada par de 1 adyacentes es cubierto por algún implicante primo en la expresión.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	E	C	–	0
B	–	E	H	(B)	1
C	A	–	(C)	F	0
D	A	(D)	–	B	1
E	A	(E)	–	B	0
F	–	D	C	(F)	0
H	A	–	(H)	B	1

Figura 9.82 Reducción después del procedimiento de particionamiento.

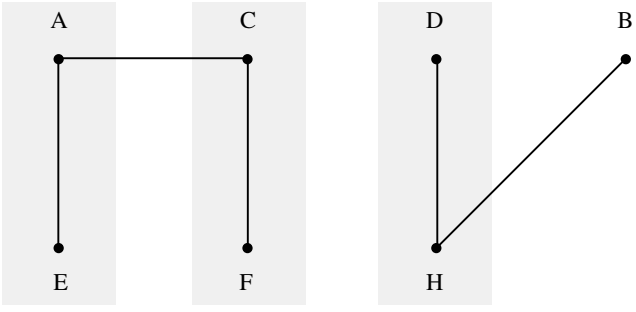


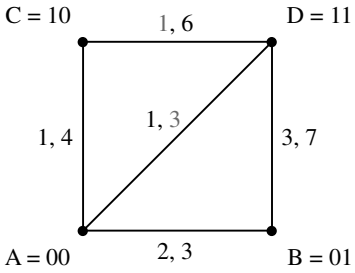
Figura 9.83 Diagrama de fusión para la tabla de flujo de la figura 9.82.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	C	B	0
B	–	A	D	(B)	1
C	A	D	(C)	(C)	0
D	A	(D)	(D)	B	1

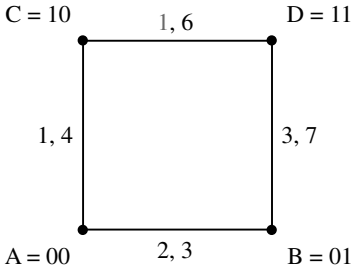
Figura 9.84 Tabla de flujo reducida para la FSM de la figura 9.82.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(1)	(2)	4	3	0
B	–	2	7	(3)	1
C	1	6	(4)	(5)	0
D	1	(6)	(7)	3	1

Figura 9.85 Tabla de flujo de la figura 9.84, reetiquetada.



a) Diagrama de transición inicial



b) Diagrama de transición aumentado

Figura 9.86 Diagramas de transición para la figura 9.85.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	C	B	0
B	–	A	D	(B)	1
C	A	D	(C)	(C)	0
D	C	(D)	(D)	B	1

a) Tabla de flujo final

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	
y_2y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	z
00	(00)	(00)	10	01	0
01	–	00	11	(01)	1
10	00	11	(10)	(10)	0
11	10	(11)	(11)	01	1

b) Tabla de excitación

Figura 9.87 Tablas de flujo y de excitación para el ejemplo 9.22.

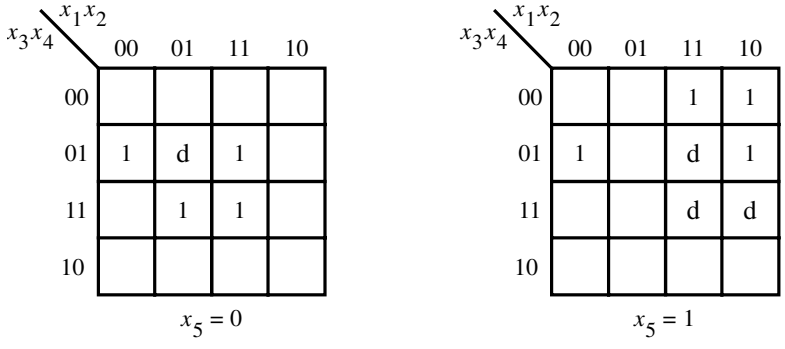


Figura 9.88 Mapa de Karnaugh para el ejemplo 9.23.

PROBLEMAS

Al final del libro se proporcionan las respuestas a los problemas marcados con asterisco.

***9.1** Derive una tabla de flujo que describa el comportamiento del circuito de la figura P9.1. Compare su solución con las tablas de la figura 9.21. ¿Hay algún parecido?

9.2 Considere el circuito de la figura P9.2. Trace un dibujo de las formas de onda para las señales C, z_1 y z_2. Suponga que C es una señal de reloj de onda cuadrada y que cada compuerta tiene un retraso de propagación Δ. Exprese el comportamiento del circuito en forma de una tabla de flujo que produzca las señales deseadas. (Consejo: utilice el modelo Mealy.)

9.3 Derive la tabla de flujo mínima que especifique el mismo comportamiento funcional que la tabla de flujo de la figura P9.3.

9.4 Derive la tabla de flujo tipo Moore mínima que especifique el mismo comportamiento funcional que la tabla de flujo de la figura P9.4.

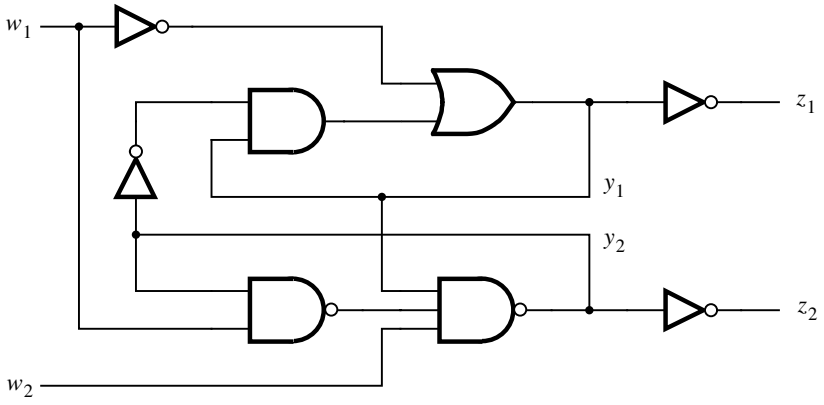


Figura P9.1 Circuito para el problema 9.1.

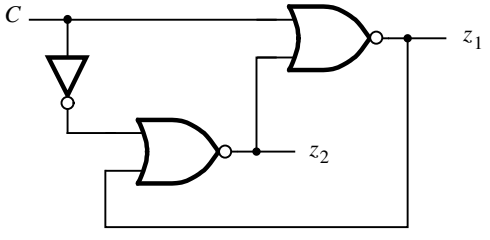


Figura P9.2 Circuito para el problema 9.2.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	P	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	M	–	(F)	–	0
G	(G)	H	I	–	0
H	J	(H)	–	–	0
I	A	–	(I)	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	–	(L)	–	1
M	(M)	N	O	–	0
N	A	(N)	–	–	1
O	A	–	(O)	–	1
P	(P)	R	S	–	0
R	T	(R)	–	–	0
S	A	–	(S)	–	1
T	(T)	U	V	–	0
U	A	(U)	–	–	1
V	A	–	(V)	–	1

Figura P9.3 Tabla de flujo para el problema 9.3.

9.5 Encuentre una tabla de asignación de estados adecuada utilizando el menor número de estados posible y derive las expresiones de estado siguiente y de salida para la tabla de flujo de la figura 9.42.

9.6 Encuentre una tabla de asignación de estados adecuada para la tabla de flujo de la figura 9.42, utilizando pares de estados equivalentes, según se explicó en el ejemplo 9.15. Derive las expresiones de estado siguiente y de salida.

9.7 Encuentre una asignación de estados para la tabla de flujo de la figura 9.42, usando la codificación de 1 activo. Derive las expresiones de estado siguiente y de salida.

***9.8** Implemente la FSM especificada en la figura 9.39, utilizando el diagrama de fusión de la figura 9.40*a*.

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	K	(B)	–	H	0
C	F	–	(C)	M	0
D	(D)	E	J	–	1
E	A	(E)	–	M	0
F	(F)	L	J	–	0
G	D	(G)	–	H	0
H	–	G	J	(H)	1
J	F	–	(J)	H	0
K	(K)	L	C	–	1
L	A	(L)	–	H	0
M	–	G	C	(M)	1

Figura P9.4 Tabla de flujo para el problema 9.4.

9.9 Encuentre una asignación de estados apropiada para la FSM definida por la tabla de flujo de la figura P9.5. Derive las expresiones de estado siguiente y de salida para la FSM utilizando esta asignación de estados.

***9.10** Encuentre una implementación de costo mínimo libre de riesgos de la función

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 11, 13, 15) + D(2, 3, 5, 10)$$

Estado presente	Estado siguiente				Salida
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	G	0
C	F	–	(C)	G	0
D	(D)	E	C	–	1
E	A	(E)	–	G	0
F	(F)	E	C	–	0
G	–	B	C	(G)	1

Figura P9.5 Tabla de flujo para el problema 9.9.

9.11 Repita el problema 9.10 para la función

$$f(x_1, \ldots, x_5) = \sum m(0, 4, 5, 24, 25, 29) + D(8, 13, 16, 21)$$

***9.12** Encuentre una implementación POS de costo mínimo y libre de riesgos de la función

$$f(x_1, \ldots, x_4) = \Pi M(0, 2, 3, 7, 10) + D(5, 13, 15)$$

9.13 Repita el problema 9.12 para la función

$$f(x_1, \ldots, x_5) = \Pi M(2, 6, 7, 25, 28, 29) + D(0, 8, 9, 10, 11, 21, 24, 26, 27, 30)$$

***9.14** Considere el circuito de la figura P9.6. ¿Exhibe algún riesgo?

9.15 Diseñe un circuito original que exhiba un riesgo dinámico.

9.16 Un mecanismo de control para una máquina expendedora acepta monedas de 5 y 10 centavos. Despacha mercancía cuando se depositan 20 centavos; no da cambio si se depositan 25 centavos. Diseñe la FSM que implemente el control requerido, con el menor número de estados posible. Encuentre una asignación de estados adecuada y derive las expresiones de estado siguiente y de salida.

***9.17** Diseñe un circuito asíncrono que satisfaga las especificaciones siguientes. El circuito tiene dos entradas: una entrada de reloj c y una entrada de control w. La salida, z, reproduce los pulsos del reloj cuando $w = 1$; de lo contrario, $z = 0$. Los pulsos que aparecen en z deben ser completos. En consecuencia, si $c = 1$ cuando w cambia de 0 a 1, entonces el circuito no producirá un pulso parcial en z sino que esperará hasta el siguiente pulso de reloj para generar $z = 1$. Si $c = 1$ cuando w cambia de 1 a 0, entonces debe generarse un pulso completo; es decir, $z = 1$ mientras $c = 1$. En la figura P9.7 se ilustra la operación deseada.

9.18 Repita el problema 9.17 pero con el cambio siguiente en la especificación. Mientras $w = 1$, la salida z debe tener sólo un pulso; si ocurren varios pulsos en c, únicamente el primero debe reproducirse en z.

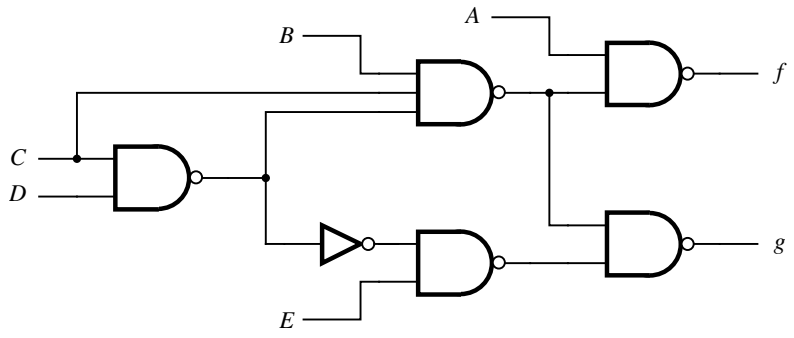


Figura P9.6 Circuito para el problema 9.14.

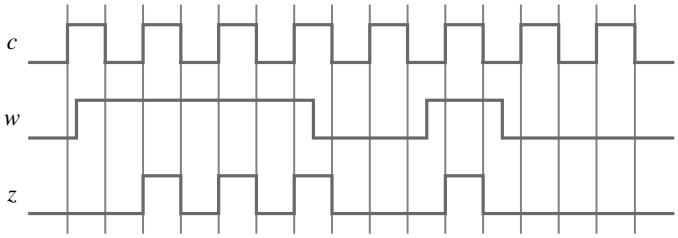


Figura P9.7 Formas de onda para el problema 9.17.

9.19 En el ejemplo 9.6 se describe un árbitro simple para dos dispositivos que compiten por un recurso compartido. Diseñe un árbitro parecido para tres dispositivos que utilicen un recurso compartido. En el caso de solicitudes simultáneas, es decir, si se ha concedido a un dispositivo acceso al recurso compartido y antes que libere su solicitud los otros dos dispositivos hacen sus solicitudes, la prioridad de los dispositivos será Dispositivo 1 > Dispositivo 2 > Dispositivo 3.

9.20 En la explicación del ejemplo 9.6 mencionamos un uso posible del elemento de exclusión mutua (ME) para evitar que las dos entradas de solicitud (request) a la FSM sean iguales a 1 al mismo tiempo. Diseñe un circuito árbitro para este caso.

9.21 En el ejemplo 9.21 diseñamos un circuito que reproduce cada segundo pulso en la entrada *w* como un pulso en la salida *z*. Diseñe un circuito similar que reproduzca cada tercer pulso.

9.22 En el ejemplo 9.22 fusionamos los estados *D* y *H* para implementar la FSM de la figura 9.82. Una alternativa era fusionar los estados *B* y *H*, de acuerdo con el diagrama de fusión de la figura 9.83. Derive una implementación usando esta opción. Construya la tabla de excitación resultante.

BIBLIOGRAFÍA

1. K. J. Breeding, *Digital Design Fundamentals* (Prentice-Hall: Englewood Cliffs, NJ, 1989).
2. F. J. Hill y G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4a. ed. (Wiley: Nueva York, 1993).
3. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
4. N. L. Pappas, *Digital Design* (West: St. Paul, MN, 1994).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 4a. ed. (West: St. Paul, MN, 1993).
6. C. J. Myers, *Asynchronous Circuit Design* (Wiley: Nueva York, 2001).

capítulo

10

Diseño de sistemas digitales

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- La habilitación de entradas para flip-flops, registros y registros de corrimiento
- Los bloques de memoria estática de acceso aleatorio (SRAM)
- Varios ejemplos de diseño de sistemas usando cartas ASM
- La sincronización del reloj
- La desviación del reloj
- La sincronización de flip-flops en el nivel del chip

En capítulos anteriores mostramos cómo diseñar muchos tipos de circuitos simples, como multiplexores, decodificadores, flip-flops, registros y contadores, los cuales pueden utilizarse como bloques de construcción. En este capítulo proporcionamos ejemplos de circuitos mucho más complejos que pueden construirse usando esos bloques como subcircuitos. Estos circuitos más grandes forman un *sistema digital.* Mostramos tanto el diseño de los circuitos para esos sistemas como la manera en que pueden describirse utilizando código de VHDL. Por razones prácticas nuestros ejemplos de sistemas digitales no serán grandes, pero las técnicas de diseño presentadas se aplican a sistemas de cualquier tamaño. Después de presentar varios ejemplos, estudiaremos algunos aspectos prácticos, por ejemplo, cómo asegurar una sincronización de flip-flops confiable en uno y en varios chips, cómo manejar las señales de entrada que no están sincronizadas con el reloj y otras cuestiones por el estilo.

Un sistema digital se compone de dos partes principales, llamadas el circuito de trayectoria de datos y el circuito de control. El *circuito de trayectoria de datos* sirve para almacenar y manipular datos, así como para transferirlos de una parte del sistema a otra. Los circuitos de trayectoria de datos comprenden bloques de construcción tales como registros, registros de corrimiento, contadores, multiplexores, decodificadores, sumadores, etcétera. El *circuito de control* vigila la buena operación del circuito de trayectoria de datos. En el capítulo 8 nos referimos a los circuitos de control como máquinas de estado finito.

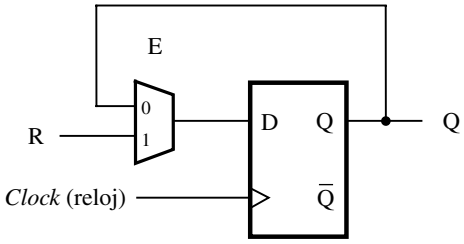
10.1 CIRCUITOS DE BLOQUE DE CONSTRUCCIÓN

Daremos varios ejemplos de sistemas digitales y mostraremos cómo diseñar sus circuitos de trayectoria de datos y de control. Los ejemplos utilizan varios de los circuitos de bloques de construcción expuestos en capítulos anteriores. Algunos de esos bloques usados en el presente capítulo se describen enseguida.

10.1.1 FLIP-FLOPS Y REGISTROS CON ENTRADAS ENABLE

En muchas aplicaciones que emplean flip-flops D es útil impedir que los datos almacenados en éstos cambien cuando ocurre un flanco de reloj activo. En la figura 7.56 mostramos cómo puede proporcionarse esta capacidad añadiendo un multiplexor al flip-flop. En la figura 10.1*a* se representa el circuito. Cuando $E = 0$, la salida del flip-flop no puede cambiar, ya que el multiplexor conecta Q a *D.* Pero si $E = 1$, entonces el multiplexor conecta la entrada *R* a *D*. En vez de utilizar el multiplexor mostrado en la figura, otra forma de implementar la función enable es ocupar una compuerta AND de dos entradas con la entrada de reloj del flip-flop. Una entrada a la compuerta AND es la señal de reloj, y la otra entrada es *E.* Entonces si se establece $E = 0$ se impide que la señal de reloj llegue a la entrada de reloj del flip-flop. Este método parece más simple que el del multiplexor, pero en la sección 10.3 mostraremos que puede causar problemas en la operación práctica. En este capítulo preferimos el método basado en multiplexor a la selección de señal de reloj con una compuerta AND.

El código de VHDL para un flip-flop D con una entrada reset asíncrona y una entrada enable se proporciona en la figura 10.1*b.* Podemos extender la capacidad enable a los registros con *n* bits al usar *n* multiplexores dos a uno controlados por *E.* El multiplexor para cada flip-flop, *i*, selecciona ya sea el bit de datos externo, R_i, o la salida del flip-flop, Q_i. El código de VHDL para un registro de *n* bits con una entrada reset asíncrona y una entrada enable se presenta en la figura 10.2.



a) Circuito

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY rege IS
    PORT ( R, Resetn, E, Clock  :IN        STD_LOGIC ;
           Q                    :BUFFER STD_LOGIC ) ;
END rege ;

ARCHITECTURE Behavior OF rege IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

b) C digo de VHDL

Figura 10.1 Un flip-flop con una entrada enable.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( R          : IN   STD_LOGIC_VECTOR(N−1 DOWNTO 0) ;
           Resetn     : IN   STD_LOGIC ;
           E, Clock   : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(N−1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= R ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 10.2 Código de VHDL para un registro de *n* bits con una entrada enable.

10.1.2 REGISTROS DE CORRIMIENTO CON ENTRADAS ENABLE

Es útil poder inhibir la operación de corrimiento en un registro de corrimiento por medio de una entrada enable, *E*. En la figura 7.19 mostramos que los registros de corrimiento pueden construirse con una capacidad de carga en paralelo, la cual se implementa con un multiplexor. En la figura 10.3 se muestra cómo es posible añadir la función enable usando un multiplexor adicional. Si la entrada de control de carga en paralelo, *L,* es 1, los flip-flops se cargan en paralelo. Pero si $L = 0$, el multiplexor adicional selecciona datos nuevos que han de cargarse en los flip-flops sólo si *E* es 1.

El código de VHDL que representa la versión de corrimiento de derecha a izquierda del circuito de la figura 10.3 se muestra en la figura 10.4. Cuando $L = 1$, el registro se carga en paralelo desde la entrada *R*. Cuando $L = 0$ y $E = 1$, los datos en el registro de corrimiento se desplazan en una dirección de derecha a izquierda.

Componentes de VHDL

Para los ejemplos presentados más adelante en este capítulo, se utilizarán varios componentes como subcircuitos. Con fines prácticos, las declaraciones del componente para estos subcircuitos se definen en el paquete de VHDL llamado *components,* que se muestra en la figura 10.5. El código para la entidad *regne* se define en la figura 10.2. El código para el *shiftlne* aparece en la figura 10.4.

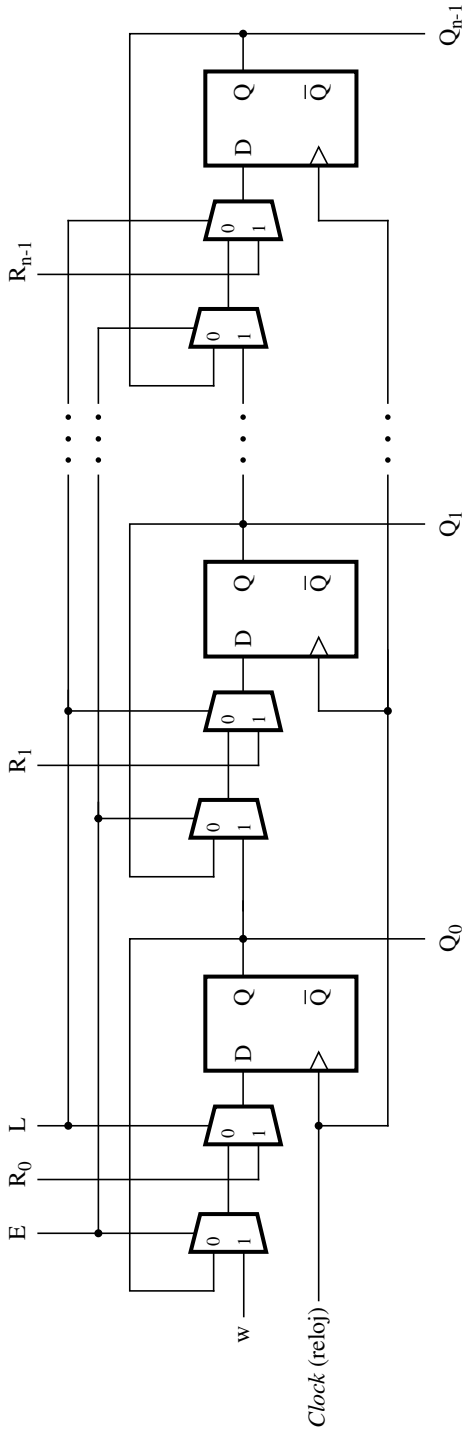


Figura 10.3 Un registro de corrimiento con entradas de control de carga en paralelo y enable.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

- - registro de corrimiento de derecha a izquierda con carga en paralelo y enable
ENTITY shiftlne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT( R       : IN     STD_LOGIC_VECTOR(N – 1 DOWNTO 0) ;
          L, E, w : IN     STD_LOGIC ;
          Clock   : IN     STD_LOGIC ;
          Q       : BUFFER STD_LOGIC_VECTOR(N – 1 DOWNTO 0) ) ;
END shiftlne ;

ARCHITECTURE Behavior OF shiftlne IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        ELSIF E = '1' THEN
            Q(0) <= w ;
            Genbits: FOR i IN 1 TO N–1 LOOP
                Q(i) <= Q(i – 1) ;
            END LOOP ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura 10.4 Código para un registro de corrimiento de derecha a izquierda con una entrada enable.

El componente *shiftrne* representa un registro de corrimiento de *n* bits con una entrada enable que desplaza a la derecha. El código se muestra en la figura 8.48. El código para las entidades *mux2to1, muxdff* y *downcnt* se da en las figuras 6.27, 7.47 y 7.54, respectivamente. La entidad *upcount* es la misma que la de la figura 7.53, con dos diferencias. Primera, se añade un parámetro GENERIC, llamado *modulus,* que especifica que los valores de conteo son 0 a *modulus*−1. Segunda, se añade una entrada enable, *E,* que impide que las salidas del contador cambien cuando $E = 0$.

10.1.3 MEMORIA ESTÁTICA DE ACCESO ALEATORIO (SRAM)

Hemos presentado varios tipos de circuitos que pueden usarse para almacenar datos. Supóngase que debemos almacenar un número grande, *m,* de elementos de datos, cada uno de los cuales se compone de *n* bits. Una posibilidad es utilizar un registro de *n* bits para cada elemento de datos. Necesitaríamos diseñar un sistema de circuitos para controlar el acceso a cada registro, tanto para cargar datos (escribir) en él como para leerlos.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS
    -- Multiplexor dos a uno
    COMPONENT mux2to1
        PORT ( w0, w1   : IN     STD_LOGIC ;
               s        : IN     STD_LOGIC ;
               f        : OUT    STD_LOGIC );
    END COMPONENT ;

    -- Flip-flop D con un multiplexor dos a uno conectado a D
    COMPONENT muxdff
        PORT ( D0, D1, Sel, E, Clock   : IN     STD_LOGIC ;
               Q                       : OUT    STD_LOGIC );
    END COMPONENT ;

    -- Registro de n bits con enable
    COMPONENT regne
        GENERIC ( N : INTEGER := 4 ) ;
        PORT ( R          : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
               Resetn     : IN     STD_LOGIC ;
               E, Clock   : IN     STD_LOGIC ;
               Q          : OUT    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;

    -- Registro de corrimiento de derecha a izquierda de n bits con carga en paralelo y enable
    COMPONENT shiftlne
        GENERIC ( N : INTEGER := 4 ) ;
        PORT ( R          : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
               L, E, w    : IN     STD_LOGIC ;
               Clock      : IN     STD_LOGIC ;
               Q          : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;
```

. . . continúa en el inciso *b*

Figura 10.5 Instrucciones de declaración de componentes para diversos bloques de construcción (inciso *a*).

Cuando m es grande, resulta poco práctico usar registros individuales para almacenar los datos. Un mejor enfoque consiste en emplear un bloque de *memoria estática de acceso aleatorio* (SRAM). Un bloque de SRAM es un arreglo bidimensional de celdas de SRAM, donde cada celda puede almacenar un bit de información. Si necesitamos almacenar m elementos con n bits cada uno podemos usar un arreglo de $m \times n$ celdas de SRAM. Las dimensiones del arreglo de SRAM se conocen como su *proporción de aspecto.*

Una celda de SRAM se parece a la celda de almacenamiento mostrada en la figura 7.3. Puesto que un bloque de SRAM puede contener un número grande de celdas de SRAM, cada

```
    -- Registro de corrimiento de izquierda a derecha de n bits con carga en paralelo y enable
    COMPONENT shiftrne
        GENERIC ( N : INTEGER := 4 ) ;
        PORT ( R        : IN     STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
               L, E, w  : IN     STD_LOGIC ;
               Clock    : IN     STD_LOGIC ;
               Q        : BUFFER STD_LOGIC_VECTOR(N – 1 DOWNTO 0) ) ;
    END COMPONENT ;

    -- Contador ascendente que cuenta de 0 a modulus−1
    COMPONENT upcount
        GENERIC ( modulus : INTEGER := 8 ) ;
        PORT ( Resetn       : IN     STD_LOGIC ;
        PORT ( Clock, E, L  : IN     STD_LOGIC ;
               R            : IN     INTEGER RANGE 0 TO modulus – 1 ;
               Q            : BUFFER INTEGER RANGE 0 TO modulus – 1 ) ;
    END COMPONENT ;

    -- Contador descendente que cuenta desde modulus−1 a 0
    COMPONENT downcnt
        GENERIC ( modulus : INTEGER := 8 ) ;
        PORT ( Clock, E, L  : IN     STD_LOGIC ;
               Q            : BUFFER INTEGER RANGE 0 TO modulus – 1 ) ;
    END COMPONENT ;
END components ;
```

Figura 10.5 Instrucciones de declaración de componentes para diversos bloques de construcción (inciso *b*).

una de éstas debe ocupar el menor espacio posible en un chip de circuito integrado. Por esta razón, la celda de almacenamiento utiliza el menor número de transistores posible. Una celda de almacenamiento popular usada en la práctica se representa en la figura 10.6. Funciona como sigue. Para almacenar datos en la celda, la entrada *Sel* se establece en 1, y el valor de los datos que van a almacenarse se coloca en la entrada *Data*. La celda de SRAM puede incluir una entrada separada para el complemento de los datos, indicada por el transistor mostrado en gris en la figura. Por simplicidad suponemos que ese transistor no está incluido en la celda. Después de esperar lo suficiente para que los datos se propaguen a través de la trayectoria de retroalimentación formada por las dos compuertas NOT, *Sel* se cambia a 0. Los datos almacenados permanecen por tanto en el ciclo de retroalimentación en forma indefinida. Un problema posible es que cuando *Sel* = 1, el valor de *Data* tal vez no sea el mismo que el que se está manejando por medio de la pequeña compuerta NOT en la trayectoria de retroalimentación. Por consiguiente, el transistor controlado por *Sel* puede intentar llevar los datos almacenados a un valor lógico mientras la salida de la pequeña compuerta NOT tiene el valor lógico opuesto. Para resolver este problema, la compuerta NOT en la trayectoria de retroalimentación se construye usando transistores pequeños (endebles), de modo que su salida pueda sobrescribirse con los datos nuevos.

Para leer los datos almacenados en la celda simplemente establecemos *Sel* en 1. En este caso el nodo *Data* no conduciría a ningún valor por el sistema de circuitos externo, de tal forma que la

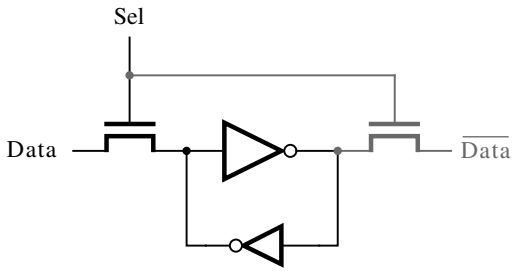


Figura 10.6 Una celda de SRAM.

celda de SRAM puede colocar los datos almacenados en este nodo. La señal *Data* se pasa por un buffer, que no se muestra en la figura, y se proporciona como una salida del bloque de SRAM.

Un bloque de SRAM contiene un arreglo de celdas de SRAM. En la figura 10.7 se muestra un arreglo con dos filas de dos celdas cada una. En cada columna del arreglo, los nodos *Data* de las celdas se conectan entre sí. Cada fila, *i*, tiene una entrada select, Sel_i, independiente, que se utiliza para leer o escribir el contenido de las celdas en esa fila. Los arreglos más grandes están formados por la conexión de más celdas a Sel_i en cada fila y por la adición de más filas. El bloque de SRAM también debe contener un sistema de circuitos que controle el acceso a cada fila del arreglo. En la figura 10.8 se representa un arreglo de $2^m \times n$ del tipo de la figura 10.7, el cual tiene un decodificador que maneja las entradas *Sel* en cada fila del arreglo. Las entradas al decodificador se llaman *Address* o *de dirección*. Este término proviene de la noción de que la ubica-

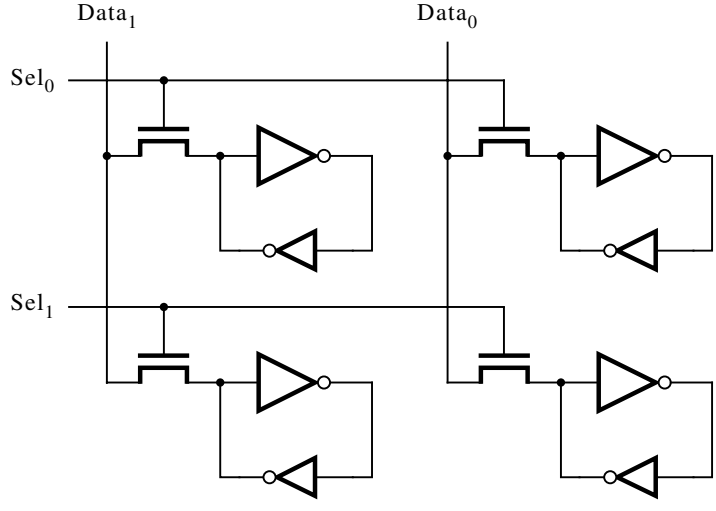


Figura 10.7 Un arreglo 2 × 2 de celdas de SRAM.

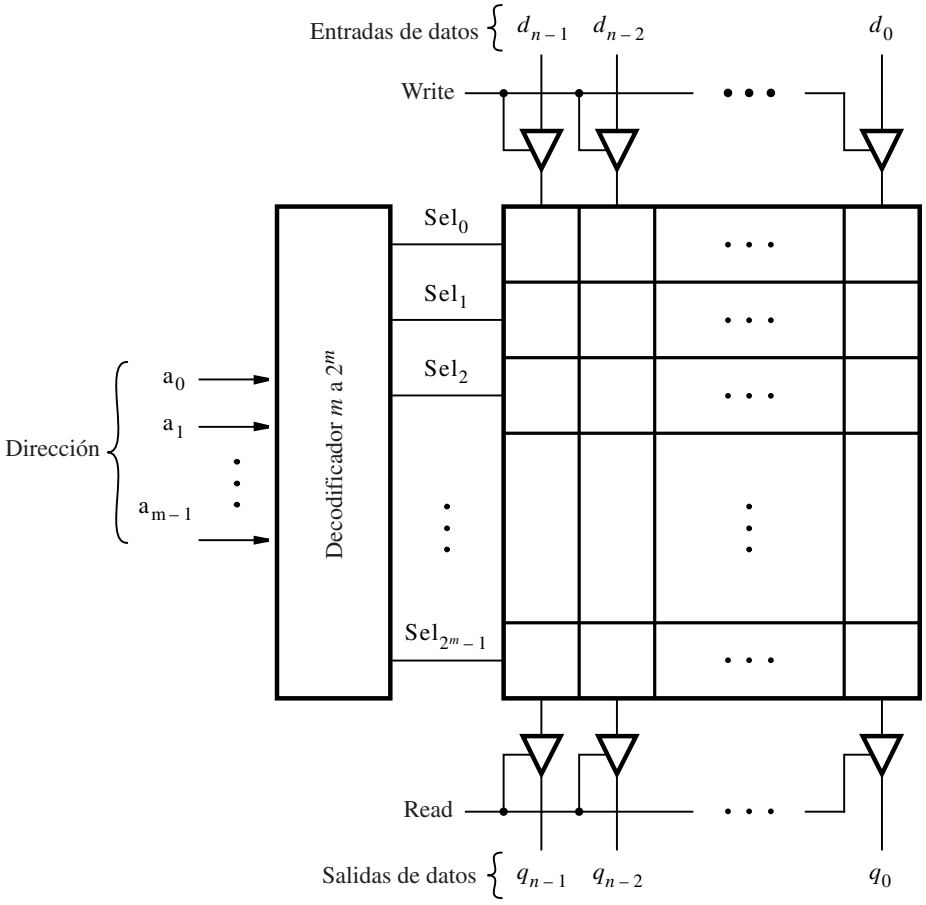


Figura 10.8 Un bloque SRAM de $2^m \times n$.

ción de una fila del arreglo puede pensarse como la "dirección" de la fila. El decodificador tiene m entradas *de dirección* y produce 2^m salidas select. Si la entrada de control *Write* es 1, entonces los bits de datos en las entradas $d_{n-1},\ldots, d_0$ se almacenan en las celdas de la fila seleccionada por las entradas *de dirección*. Si la entrada de control *Read* es 1, entonces los datos almacenados en la fila seleccionada por las entradas *de dirección* aparecen en las salidas $q_{n-1},\ldots, q_0$. En muchas aplicaciones prácticas las entradas y las salidas de datos están conectadas entre sí. Por tanto, las entradas *Write* y *Read* nunca deben tener el valor 1 al mismo tiempo.

El diseño de bloques de memoria ha sido sujeto de una investigación y desarrollo intensivos. Sólo hemos descrito la operación básica de un tipo de bloque de memoria. El lector puede remitirse a libros de organización de computadoras para más información [1, 2].

10.1.4 BLOQUES SRAM EN PLD

Algunos PLD contienen bloques de SRAM que pueden usarse como parte de los circuitos implementados en los chips. Un chip popular tiene una serie de bloques de SRAM, cada uno de los cuales contiene 4096 celdas de SRAM. Esos bloques pueden configurarse para proporcionar diferentes proporciones de aspecto, según las necesidades del diseño que se está implementando. Pueden producirse proporciones de aspecto desde 512×8 hasta $4\,096 \times 1$ usando un solo bloque de SRAM, y pueden combinarse varios bloques para formar arreglos de memoria más grandes. Para incluir bloques de SRAM en un circuito, los diseñadores emplean módulos preconstruidos que se proporcionan en una biblioteca como parte de las herramientas CAD, o escriben código de VHDL desde el cual las herramientas de síntesis pueden inferir los bloques de memoria.

10.2 EJEMPLOS DE DISEÑO

En la sección 8.10 presentamos las cartas ASM (ASM, *algorithmic state machine*) y mostramos cómo pueden utilizarse para describir máquinas de estado finito. Las cartas ASM también pueden usarse para describir sistemas digitales que incluyen tanto los circuitos de trayectoria de datos como los de control. Ilustraremos con varios ejemplos cómo se emplean esas cartas para facilitar el diseño de los sistemas digitales.

10.2.1 UN CIRCUITO DE CONTEO DE BITS

Supóngase que deseamos contar el número de bits en un registro, *A,* que tienen el valor 1. En la figura 10.9 se muestra un seudocódigo para un procedimiento paso por paso, o *algoritmo*, que se utiliza para realizar la tarea requerida. Este código supone que *A* está almacenado en un registro que puede desplazar su contenido en dirección de izquierda a derecha. La respuesta producida por el algoritmo se almacena en la variable llamada *B.* El algoritmo termina cuando *A* no contiene más unos (1), es decir, cuando $A = 0$. En cada iteración del ciclo while, si el bit menos significativo (LSB) de *A* es 1, entonces *B* se incrementa en 1; de lo contrario, *B* no sufre cambios. *A* se desplaza un bit a la derecha al final de cada iteración del ciclo.

En la figura 10.10 se presenta una carta ASM que representa el algoritmo de la figura 10.9. El cuadro de estado para el estado inicial, *S*1*,* especifica que *B* se inicializa en 0. Suponemos

```
B = 0 ;
while A ≠ 0 do
      if a0 = 1 then
            B = B + 1 ;
      End if ;
      Right-shift A ;
end while ;
```

Figura 10.9 Seudocódigo para el contador de bits.

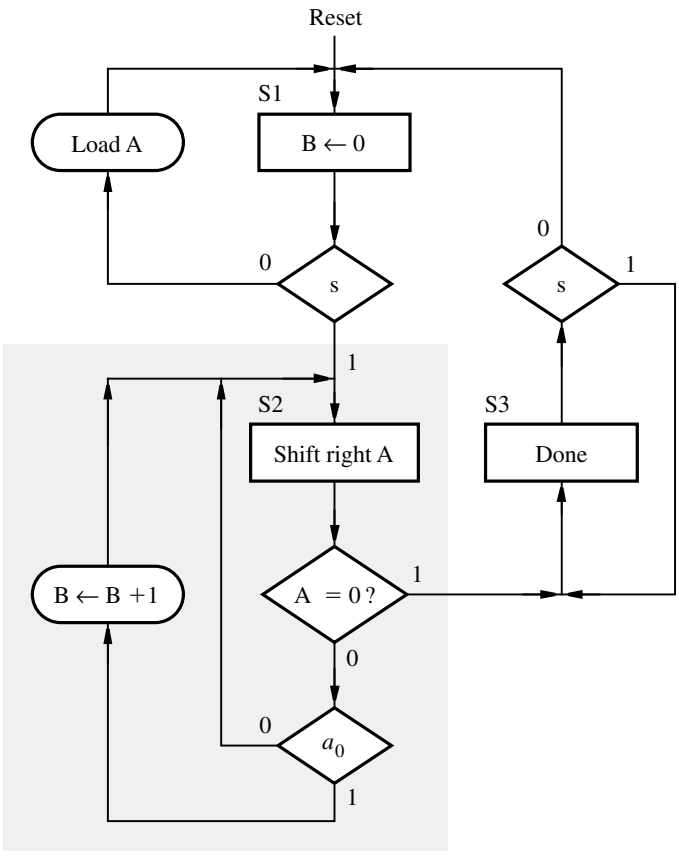


Figura 10.10 Carta ASM para el seudocódigo de la figura 10.9.

que existe una señal de entrada, *s,* que se utiliza para indicar cuándo deben cargarse en *A* los datos que van a procesarse, de modo que la máquina pueda ponerse en marcha. El cuadro de decisión etiquetado con una *s* estipula que la máquina permanece en el estado *S*1 mientras $s = 0$. El cuadro de salida condicional etiquetado con las palabras *Load A* indica que *A* se carga desde entradas de datos externas si $s = 0$ en el estado *S*1.

Cuando *s* se vuelve 1, la máquina cambia al estado *S*2. El cuadro de decisión debajo del cuadro de estado para *S*2 revisa si $A = 0$. Si es así, la operación de conteo de bits está completa; por consiguiente, la máquina debe cambiar al estado *S*3. De lo contrario, la FSM permanece en el estado *S*2. El cuadro de decisión en la parte inferior de la carta revisa el valor de a_0. Si $a_0 = 1$, *B* se incrementa, lo cual se indica en la carta como $B \leftarrow B + 1$. Si $a_0 = 0$, entonces *B* no cambia. En el estado *S*3, *B* contiene el resultado, que es el número de bits en *A* que eran 1. Una señal de salida, *Done,* se establece en 1 para indicar que el algoritmo ha terminado; la FSM se queda en *S*3 hasta que regresa a 0.

10.2.2 INFORMACIÓN DE SINCRONIZACIÓN ESBOZADA EN LA CARTA ASM

En la sección 8.10 dijimos que las cartas ASM son similares a los diagramas de flujo tradicionales, salvo porque incluyen información sobre la sincronización. Podemos usar el ejemplo del conteo de bits para ilustrar este concepto. Considérese el bloque ASM para el estado *S*2, el cual está sombreado en la figura 10.10. En un diagrama de flujo tradicional, cuando se entra en el estado *S*2, el valor de *A* primero se desplazaría a la derecha. Entonces examinaríamos el valor de *A* y si su LSB es 1, de inmediato añadiríamos 1 a *B*. Pero como la carta ASM representa un circuito secuencial, los cambios en *A* y en *B*, que representan las salidas de los flip-flops, ocurren después del flanco activo del reloj. La misma señal de reloj que controla los cambios en el estado de la máquina también controla los cambios en *A* y en *B*. Por ende, en el estado *S*2 el cuadro de decisión que prueba si $A = 0$, así como el cuadro que revisa el valor de a_0, examina los bits en *A* antes que se desplacen. Si $A = 0$, entonces la FSM cambiará al estado *S*3 en el siguiente flanco del reloj (este flanco del reloj también desplaza *A*, lo que no tiene ningún efecto porque *A* ya es 0 en este caso). Por otra parte, si $A \neq 0$, entonces la FSM no cambia a *S*3, sino que permanece en *S*2. Al mismo tiempo, *A* aún se recorre y *B* se incrementa si a_0 tiene el valor 1. Estos aspectos de la sincronización se ilustran en la figura 10.14, la cual representa el resultado de una simulación para un circuito que implementa la carta ASM. Mostramos cómo se diseña el circuito en la explicación siguiente.

Circuito de trayectoria de datos

Al examinar la carta ASM para el circuito de conteo de bits podemos inferir el tipo de elementos de circuito requeridos para implementar su trayectoria de datos. Necesitamos un registro de corrimiento que realice el desplazamiento de izquierda a derecha para implementar *A*. Debe contar con la capacidad de carga en paralelo debido al cuadro de salida condicional en el estado *S*1 que carga los datos en el registro. También se requiere una entrada enable porque el desplazamiento debe ocurrir únicamente en el estado *S*2. Se precisa un contador para *B*, y necesita la capacidad de carga en paralelo para inicializar el conteo a 0 en el estado *S*1. No es recomendable basarse en la entrada reset del contador para borrar *B* a 0 en el estado *S*1. En la práctica, la señal reset se usa en un sistema digital sólo con dos propósitos: inicializar el circuito cuando se energiza por vez primera o para recuperarse de un error. La máquina cambia del estado *S*3 a *S*1 como resultado de $s = 0$; por consiguiente no debemos suponer que la señal reset se emplea para borrar el contador.

El circuito de trayectoria de datos se representa en la figura 10.11. La entrada serial al registro de corrimiento, *w*, está conectada a 0 porque no se necesita. Las entradas load y enable en el registro de corrimiento están manejadas por las señales *LA* y *EA*. Las entradas en paralelo al registro de corrimiento se llaman *Data*, y su salida en paralelo es *A*. Se utiliza una compuerta NOR de *n* entradas para probar si $A = 0$. La salida de esta compuerta, *z*, es 1 cuando $A = 0$. Obsérvese que la figura indica la compuerta NOR con *n* entradas al mostrar una conexión de una sola entrada a la compuerta, con la etiqueta *n* adscrita a ella. El contador tiene $log_2(n)$ bits, con entradas en paralelo conectadas a 0 y salidas en paralelo llamadas *B*. También tiene una entrada de carga en paralelo *LB* y una señal de control enable *EB*.

Circuito de control

Por conveniencia podemos trazar una segunda carta ASM que represente sólo la FSM necesaria para el circuito de control, como se muestra en la figura 10.12. La FSM tiene las entradas *s*, a_0 y *z*, y genera las salidas *EA*, *LB*, *EB* y *Done*. En el estado *S*1, *LB* se activa, de modo que un 0 se carga en paralelo en el contador. Nótese que para las señales de control, como *LB*, en vez de escribir $LB = 1$ simplemente escribimos *LB* para indicar que la señal se valida. Suponemos que el sistema de circuitos externo lleva *LA* a 1 cuando se presentan datos válidos en las entradas en

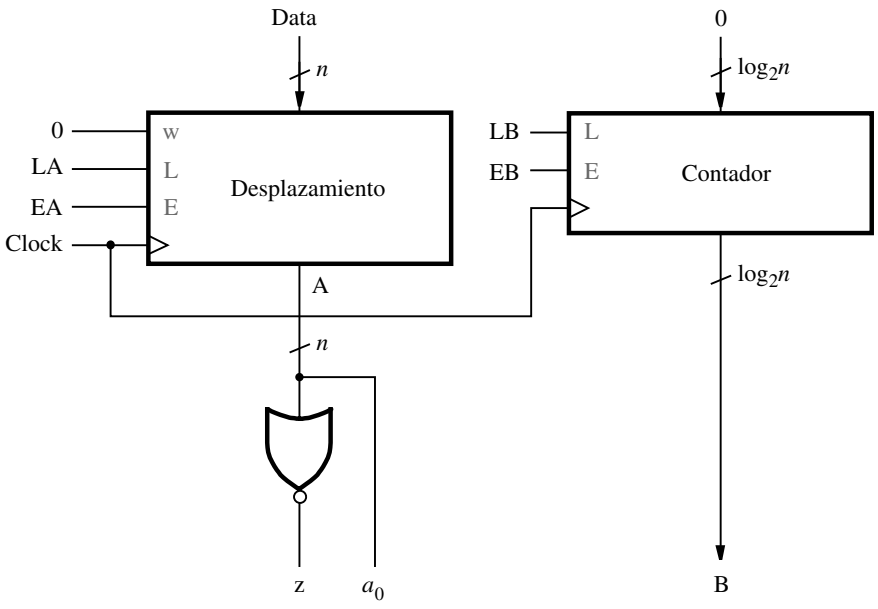


Figura 10.11 Trayectoria de datos para la carta ASM de la figura 10.10.

paralelo del registro de corrimiento, de modo que el contenido del registro se inicializa antes que *s* cambie a 1. En el estado *S*2, *EA* se activa para provocar una operación de desplazamiento y la habilitación del conteo para *B* se activa sólo si $a_0 = 1$.

Código de VHDL

El circuito de conteo de bits puede describirse en código de VHDL como se muestra en la figura 10.13. Hemos elegido definir *A* como una señal de ocho bits STD_LOGIC_VECTOR y *B,* como una señal de número entero. La carta ASM de la figura 10.12 puede traducirse directamente en un código que describe el circuito de control requerido. La señal llamada *y* se usa para representar los flip-flops de estado y el proceso llamado *FSM_transitions,* al principio del cuerpo de arquitectura, especifica las transiciones de estado. El proceso etiquetado *FSM_outputs* especifica las salidas generadas en cada estado. Un valor predeterminado se especifica al principio de este proceso para todas las señales de salida, y los valores de salida individuales se especifican en la instrucción case.

El proceso etiquetado *upcount* define el contador ascendente que implementa *B.* El registro de corrimiento para *A* se consigna al final del código y la señal *z* se define mediante una asignación de señal condicional. Implementamos el código de la figura 10.13 en un chip y realizamos una simulación de sincronización. En la figura 10.14 se muestran los resultados de la simulación para $A = 00111011$. Después de que el circuito se inicializa, la señal de entrada *LA* se establece en 1, y los datos deseados, $(3B)_{16}$, se colocan en las entradas *Data*. Cuando *s* cambia a 1, el siguiente flanco activo del reloj hace que la FSM cambie al estado *S*2. En este estado cada flanco activo del reloj incrementa *B* si a_0 es 1, y recorre *A.* Cuando $A = 0$, el siguiente flanco activo del reloj ocasiona que la FSM cambie al estado S3, donde *Done* se establece en 1 y *B* tiene el resultado correcto, $B = 5$. Para revisar más rigurosamente que el circuito se diseñó de manera correcta debemos probar diferentes valores de datos de entrada.

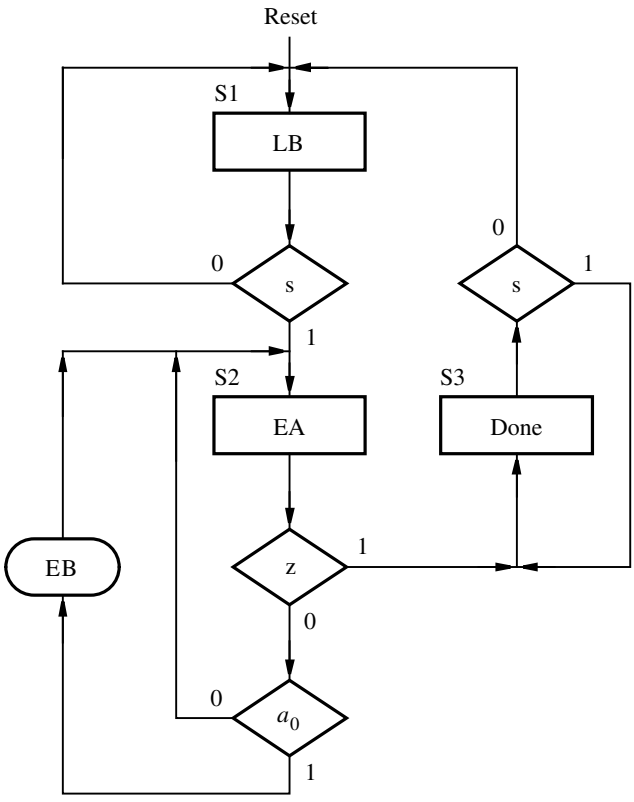


Figura 10.12 Carta ASM para el circuito de control del contador.

10.2.3 MULTIPLICADOR DE CORRIMIENTO Y SUMA

En la figura 5.32 presentamos un circuito que multiplica dos números binarios de *n* bits sin signo. El circuito utiliza dos arreglos bidimensionales de subcircuitos idénticos, cada uno de los cuales contiene un sumador completo y una compuerta AND. Para valores grandes de *n,* este enfoque tal vez no sea apropiado debido al gran número de compuertas requeridas. Otro método consiste en usar un registro de corrimiento junto con un sumador para implementar el método tradicional de multiplicación que se hace "a mano". En la figura 10.15*a* se ilustra este proceso manual de multiplicar dos números binarios. El producto se forma con una serie de operaciones de adición. Para cada bit *i* en el multiplicador que es 1, sumamos al producto el valor del multiplicando desplazado a la izquierda *i* veces. Este algoritmo puede describirse en seudocódigo como se muestra en la figura 10.15*b*, donde *A* es el multiplicando, *B* el multiplicador y *P* el producto.

Una carta ASM que representa el algoritmo en la figura 10.15*b* se presenta en la figura 10.16. Suponemos que se emplea una entrada *s* a fin de controlar cuándo la máquina comienza el proceso de multiplicación. Mientras *s* sea 0, la máquina permanece en el estado *S*1 y los datos para *A* y *B* pueden cargarse desde entradas externas. En el estado *S2* probamos el valor LSB de

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY work ;
USE work.components.shiftrne ;

ENTITY bitcount IS
    PORT( Clock, Resetn  : IN      STD_LOGIC ;
          LA, s          : IN      STD_LOGIC ;
          Data           : IN      STD_LOGIC_VECTOR(7 DOWNTO 0) ;
          B              : BUFFER  INTEGER RANGE 0 to 8 ;
          Done           : OUT     STD_LOGIC ) ;
END bitcount ;

ARCHITECTURE Behavior OF bitcount IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL A : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL z, EA, LB, EB, low : STD_LOGIC ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

. . . continúa en el inciso *b*

Figura 10.13 Código de VHDL para el circuito de conteo de bits (inciso *a*).

B, y si es 1, sumamos *A* a *P*. De lo contrario, *P* no cambia. La máquina se mueve al estado *S*3 cuando *B* contiene 0, ya que *P* tiene el producto final en este caso. Por cada ciclo de reloj en el que la máquina se halla en el estado *S*2, recorremos el valor de *A* a la izquierda, según se especifica en el seudocódigo de la figura 10.15*b*. Desplazamos el contenido de *B* a la derecha de modo que en cada ciclo de reloj b_0 pueda utilizarse para decidir si *A* debe añadirse a *P* o no.

```
    FSM_outputs: PROCESS ( y, A(0) )
    BEGIN
        EA <= '0' ; LB <= '0' ; EB <= '0' ; Done <= '0' ;
        CASE y IS
            WHEN S1 =>
                LB <= '1' ;
            WHEN S2 =>
                EA <= '1' ;
                IF A(0) = '1' THEN EB <= '1' ; ELSE EB <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;

    -- El circuito de trayectoria de datos se describe enseguida
    upcount: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            B <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF LB = '1' THEN
                B <= 0 ;
            ELSIF EB = '1' THEN
                B <= B + 1 ;
            END IF ;
        END IF;
    END PROCESS;

    low <= '0' ;
    ShiftA: shiftrne GENERIC MAP ( N => 8 )
        PORT MAP ( Data, LA, EA, low, Clock, A ) ;
    z <= '1' WHEN A = "00000000" ELSE '0' ;
END Behavior ;
```

Figura 10.13 Código de VHDL para el circuito de conteo de bits (inciso *b*).

Circuito de trayectoria de datos

Ahora podemos definir el circuito de trayectoria de datos. Para implementar A necesitamos un registro de corrimiento de derecha a izquierda que tenga $2n$ bits. Se precisa también un registro de $2n$ bits para P, y ha de tener una entrada enable porque la asignación $P \leftarrow P + A$ en el estado $S2$ está dentro de un cuadro de salida condicional. Se necesita un sumador de $2n$ bits para producir $P + A$. Nótese que P está cargado con 0 en el estado $S1$, y se carga desde la salida del sumador en el estado $S2$. No podemos suponer que la entrada reset se usa para borrar P, ya que la

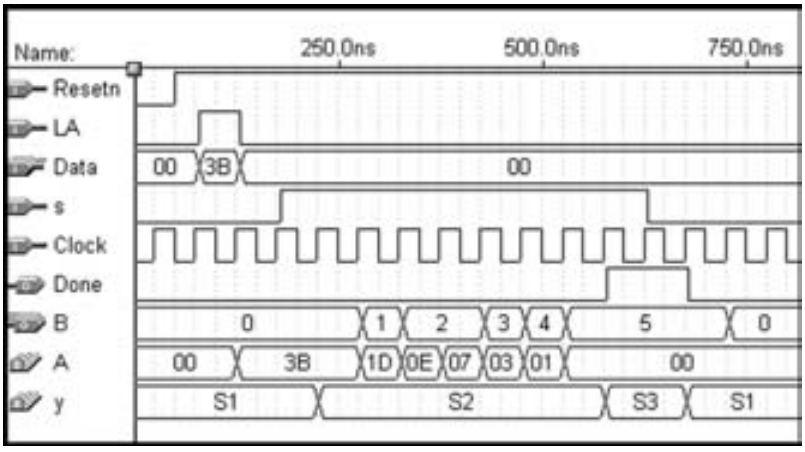


Figura 10.14 Resultados de la simulación para el circuito de conteo de bits.

Decimal		Binario	
13		1 1 0 1	Multiplicando
× 11		× 1 0 1 1	Multiplicador
13		1 1 0 1	
13		1 1 0 1	
143		0 0 0 0	
		1 1 0 1	
		1 0 0 0 1 1 1 1	Producto

a) Método manual

```
P = 0 ;
for i = 0 to n − 1 do
    if b_i = 1 then
        P = P + A ;
    end if ;
    Left-shift A ;
end for ;
```

b) Seudocódigo

Figura 10.15 Un algoritmo para la multiplicación.

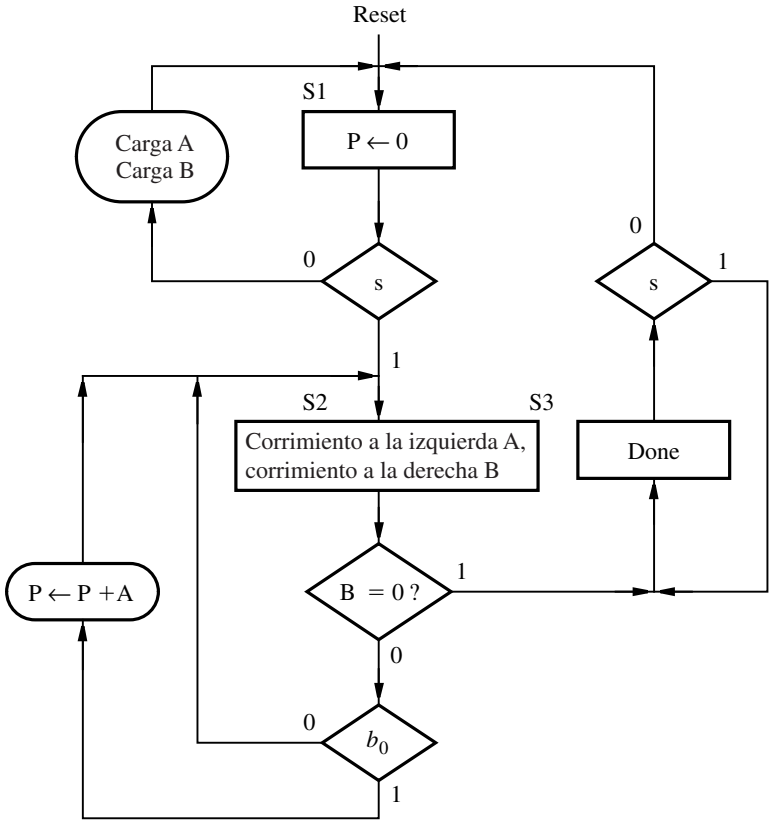


Figura 10.16 Carta ASM para el multiplicador.

máquina cambia del estado *S*3 al estado *S*1 otra vez con base en la entrada *s*, no en la reset. Por consiguiente, se necesita un multiplexor dos a uno para cada entrada a *P,* a fin de seleccionar ya sea 0 o el bit de suma apropiado del sumador. Se requiere también un registro de corrimiento de *n* bits de izquierda a derecha para *B,* y una compuerta NOR de *n* entradas puede utilizarse para probar si $B = 0$.

En la figura 10.17 se muestra el circuito de trayectoria de datos y se etiquetan las señales de control para el registro de corrimiento. Los datos de entrada para el registro de corrimiento que almacena *A* se llaman *DataA.* Puesto que el registro de corrimiento tiene 2*n* bits, las *n* entradas de datos más significativas están conectadas a 0. Un solo símbolo multiplexor se muestra conectado al registro que aloja a *P.* Ese símbolo representa 2*n* multiplexores de dos a uno controlados por la señal *Psel*.

Circuito de control

Una carta ASM que representa sólo las señales de control necesarias para el multiplicador se proporciona en la figura 10.18. En el estado *S*1, *Psel* se establece en 0 y *EP* se activa, de modo que el registro *P* se borra. Cuando $s = 0$, los datos en paralelo pueden cargarse en los registros de corrimiento *A* y *B* mediante un circuito externo que controla sus entradas de carga en paralelo *LA* y *LB.* Cuando $s = 1$, la máquina cambia al estado *S*2, donde *Psel* se establece en 1 y se habilita

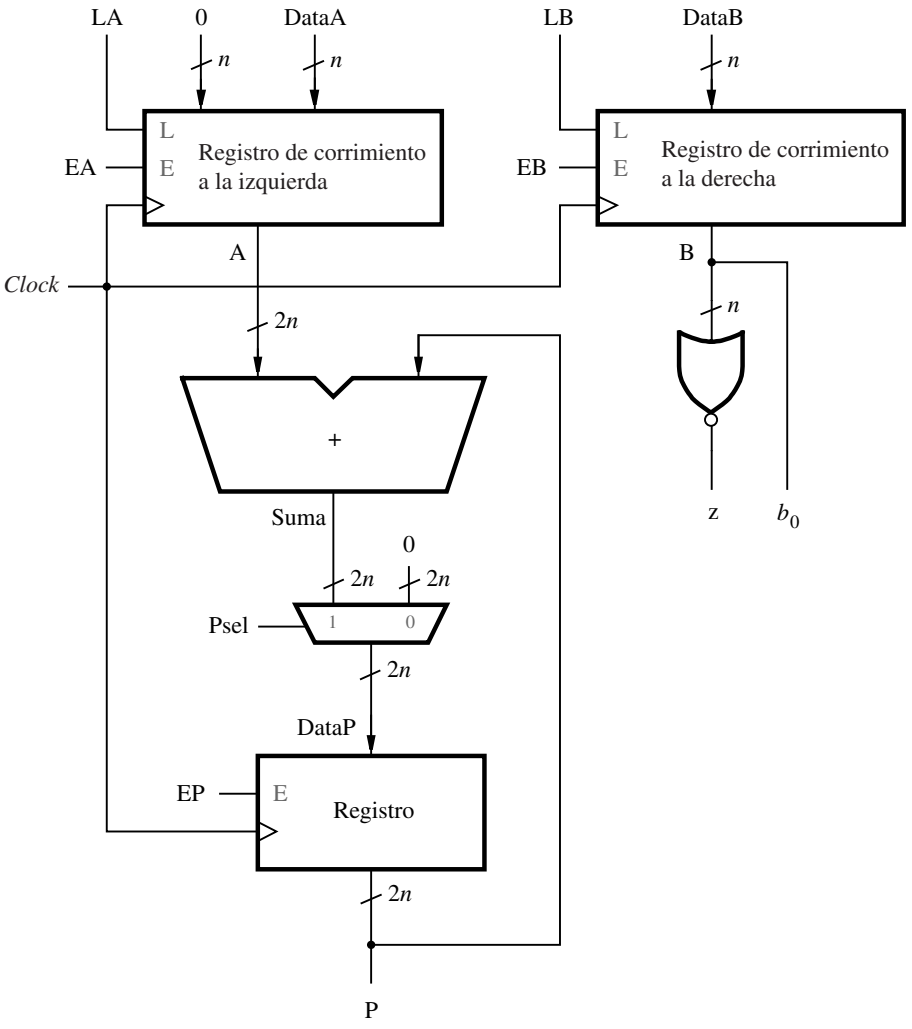


Figura 10.17 Circuito de trayectoria de datos para el multiplicador.

el corrimiento de *A* y de *B*. Si $b_0 = 1$, se activa la entrada enable para *P*. La máquina cambia al estado *S*3 cuando $z = 1$, permanece en él y establece *Done* al valor 1 mientras $s = 1$.

Código de VHDL

El código de VHDL para el multiplicador se presenta en la figura 10.19. El parámetro genérico *N* establece el número de bits en *A* y *B.* Como algunos registros miden *2n* bits, un segundo parámetro genérico *NN* se define para representar $2 \times N$. Al cambiar el valor de los parámetros genéricos, el código puede utilizarse para números de cualquier tamaño. Los procesos etiquetados *FSM_transitions* y *FSM_outputs* definen las transiciones de estado y las salidas generadas,

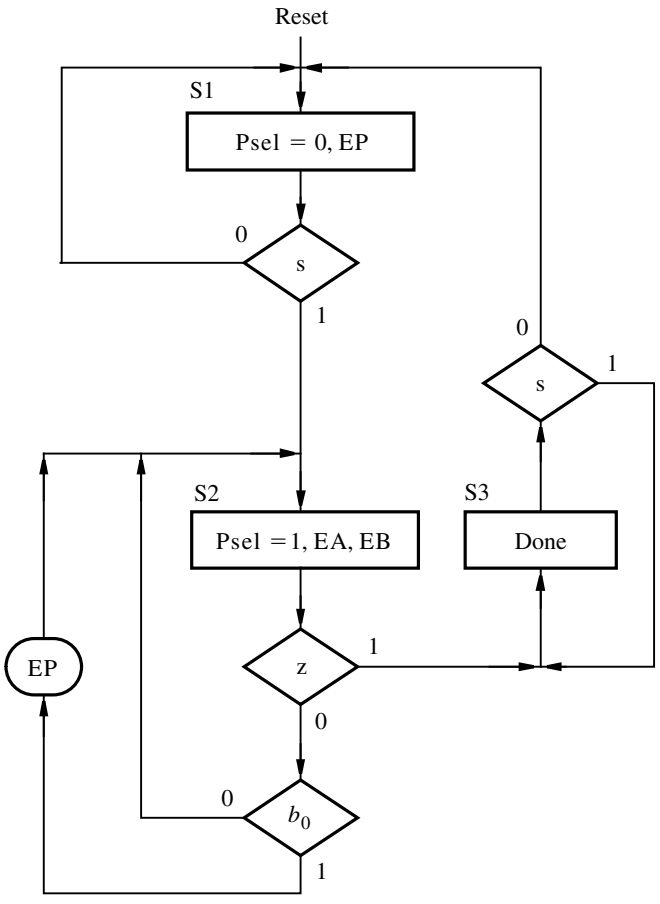


Figura 10.18 Carta ASM del circuito de control para el multiplicador.

respectivamente, en el circuito de control. La entrada de datos en paralelo en el registro de corrimiento *A* mide 2*N* bits, pero *DataA* mide sólo *N* bits. La señal *N_Zeros* se utiliza para generar *n* bits con valor de cero, y en la señal *Ain* se añaden estos bits con *DataA* para cargarlos en el registro de corrimiento. El multiplexor necesario para el registro *P* se define con una instrucción FOR GENERATE que instancia 2*N* multiplexores dos a uno. En la figura 10.20 se proporciona el resultado de una simulación para el circuito generado con base en el código. Después que el circuito se inicializa, *LA* y *LB* se establecen en 1, y los números que se multiplicarán se colocan en las entradas *DataA* y *DataB*. Luego que *s* se establece en 1, la FSM (*y*) cambia al estado *S*2, donde permanece hasta que $B = 0$. Para cada ciclo de reloj en el estado *S*2, *A* se recorre a la izquierda y *B* a la derecha. En tres de los ciclos de reloj en el estado *S*2 el contenido de *A* se suma a *P,* lo cual corresponde a los tres bits en *B* que tienen el valor 1. Cuando $B = 0$, la FSM cambia al estado *S*3 y *P* contiene el producto correcto, el cual es $(64)_{16} \times (19)_{16} = (9C4)_{16}$. El equivalente decimal de este resultado es $100 \times 25 = 2500$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY multiply IS
    GENERIC ( N : INTEGER := 8; NN : INTEGER := 16 ) ;
    PORT ( Clock      : IN     STD_LOGIC ;
           Resetn     : IN     STD_LOGIC ;
           LA, LB, s  : IN     STD_LOGIC ;
           DataA      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           DataB      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           P          : BUFFER STD_LOGIC_VECTOR(NN-1 DOWNTO 0) ;
           Done       : OUT    STD_LOGIC ) ;
END multiply ;

ARCHITECTURE Behavior OF multiply IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL Psel, z, EA, EB, EP, Zero : STD_LOGIC ;
    SIGNAL B, N_Zeros : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL A, Ain, DataP, Sum : STD_LOGIC_VECTOR(NN-1 DOWNTO 0) ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

. . . continúa en el inciso *b*

Figura 10.19 Código de VHDL para el circuito multiplicador (inciso *a*).

```
    FSM_outputs: PROCESS ( y, s, B(0) )
    BEGIN
        EP <= '0' ; EA <= '0' ; EB <= '0' ; Done <= '0' ; Psel <= '0';
        CASE y IS
            WHEN S1 =>
                EP <= '1' ;
            WHEN S2 =>
                EA <= '1' ; EB <= '1' ; Psel <= '1' ;
                IF B(0) = '1' THEN EP <= '1' ; ELSE EP <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;

    -- Define el circuito de trayectoria de datos
    Zero <= '0' ;
    N_Zeros <= (OTHERS => '0' ) ;
    Ain <= N_Zeros & DataA ;
    ShiftA: shiftlne GENERIC MAP ( N => NN )
        PORT MAP ( Ain, LA, EA, Zero, Clock, A ) ;
    ShiftB: shiftrne GENERIC MAP ( N => N )
        PORT MAP ( DataB, LB, EB, Zero, Clock, B ) ;
    z <= '1' WHEN B = N_Zeros ELSE '0' ;
    Sum <= A + P ;
    -- Define los 2n multiplexores dos a uno para DataP
    GenMUX: FOR i IN 0 TO NN-1 GENERATE
        Muxi: mux2to1 PORT MAP ( Zero, Sum(i), Psel, DataP(i) ) ;
    END GENERATE;
    RegP: regne GENERIC MAP ( N => NN )
        PORT MAP ( DataP, Resetn, EP, Clock, P ) ;
END Behavior ;
```

Figura 10.19 Código de VHDL para el circuito multiplicador (inciso *b*).

El número de ciclos de reloj que requiere el circuito para generar el producto final está determinado por el dígito más a la izquierda de *B,* que sea 1. Es posible reducir el número de ciclos de reloj necesarios mediante registros de corrimientos más complejos para *A* y *B.* Si los dos bits en el extremo derecho en *B* son 0, entonces *A* y *B* podrían desplazarse dos posiciones de bit en un ciclo de reloj. De forma similar, si los tres dígitos inferiores en *B* son 0, entonces puede hacerse un corrimiento de tres posiciones de bit y así por el estilo. Un registro de corrimiento que puede desplazar múltiples posiciones de bit de una sola vez puede construirse usando un *registro de desplazamiento cíclico*. Dejamos como ejercicio para el lector modificar el multiplicador para utilizar un registro de desplazamiento cíclico.

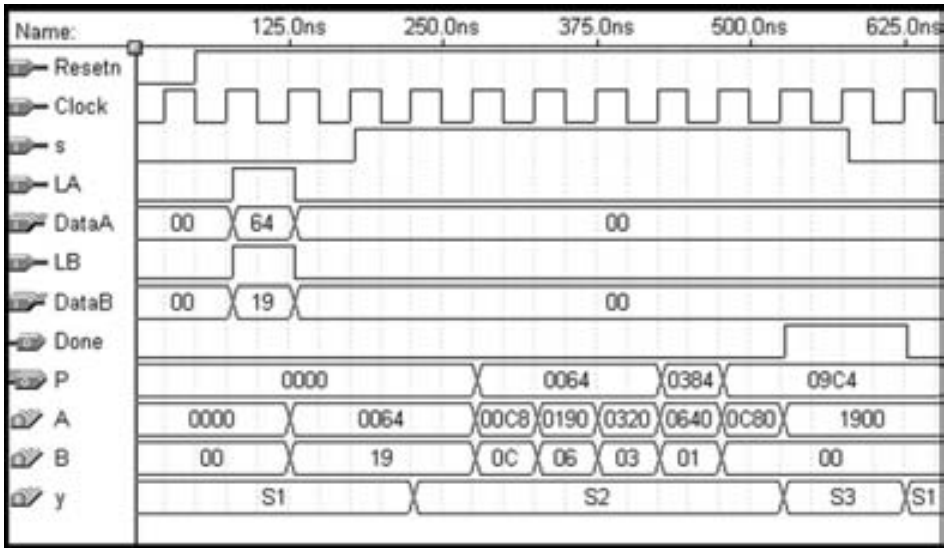


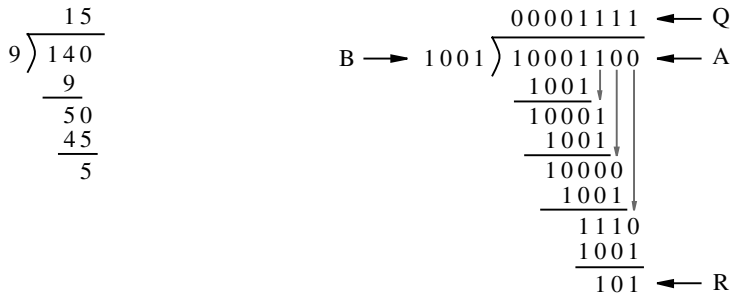
Figura 10.20 Resultados de la simulación para el circuito multiplicador.

10.2.4 DIVISOR

El ejemplo anterior implementa el método tradicional de realizar la multiplicación a mano. En este ejemplo diseñaremos un circuito que implementa la división tradicional en escritura normal. En la figura 10.21*a* se ofrece un ejemplo de división en escritura normal. El primer paso es tratar de dividir el divisor 9 entre el primer dígito del dividendo, 1, lo cual no es posible. Por ello tratamos de dividir 9 entre 14, y determinar que 1 es el primer dígito en el cociente. Realizamos la resta $14 - 9 = 5$, tomamos el último dígito del dividendo para formar 50, y luego determinamos que el siguiente dígito en el cociente es 5. El residuo es $50 - 45 = 5$, y el cociente es 15. El uso de números binarios, como se ilustra en la figura 10.21*b*, supone el mismo proceso, con la simplificación de que cada dígito del cociente puede ser únicamente 0 o 1.

Dados los números de n bits sin signo A y B, queremos diseñar un circuito que produzca dos salidas de n bits Q y R, donde Q es el cociente A/B y R es el residuo. El procedimiento ilustrado en la figura 10.21*b* puede implementarse desplazando los dígitos en A a la izquierda, uno a la vez, en un registro de corrimiento R. Después de cada operación de corrimiento, comparamos R con B. Si $R \geq B$, se coloca un 1 en la posición de bit apropiada en el cociente y B se resta de R. De lo contrario, se coloca un 0 en el cociente. Este algoritmo se describe utilizando el seudocódigo de la figura 10.21*c*. La notación $R||A$ se usa para representar un registro de corrimiento de $2n$ bits formado utilizando R como los n bits en el extremo izquierdo y A como los n bits en el extremo derecho.

El seudocódigo para el multiplicador de la figura 10.15*b* examina un dígito, b_i, en cada iteración de ciclo. En la carta ASM de la figura 10.16, desplazamos B a la derecha de modo que b_0 siempre contiene el digito requerido. De forma similar, en el seudocódigo de la división larga, cada iteración de ciclo da como resultado que se establezca un dígito q_i ya sea en 1 o en 0. Una



a) Un ejemplo con números decimales

b) Un ejemplo con números binarios

```
R = 0 ;
for i = 0 to n – 1 do
      Left-shift R||A ;
      if R ≥ B then
            q_i = 1 ;
            R = R − B ;
      else
            q_i = 0 ;
      end if ;
end for ;
```

c) Seudocódigo

Figura 10.21 Un algoritmo para la división.

manera sencilla de lograr esto es recorrer 1 o 0 en el bit menos significativo de Q en cada iteración de ciclo. Una carta ASM que representa el circuito divisor se muestra en la figura 10.22. La señal C representa un contador que se inicializa en $n - 1$ en el estado inicial $S1$. En el estado $S2$, tanto R como A se desplazan a la izquierda y luego, en el estado $S3$, B se resta de R si $R \geq B$. La máquina cambia al estado $S4$ cuando $C = 0$.

Circuito de trayectoria de datos

Necesitamos registros de corrimiento de n bits que desplacen de derecha a izquierda para A, R y Q. Se requiere un registro de n bits para B y un restador para producir $R - B$. Podemos usar un módulo sumador en el que el acarreo de entrada se establezca en 1 y B se complemente. El acarreo de salida, c_{out}, de este módulo tiene el valor de 1 si la condición $R \geq B$ es verdadera. Por consiguiente, el acarreo de salida puede estar conectado a la entrada serial del registro de corrimiento que almacena Q, de modo que éste se desplace hacia Q en el estado $S3$. Como R se carga con 0 en el estado $S1$ y desde las salidas del sumador en el estado $S3$, se necesita un multiplexor

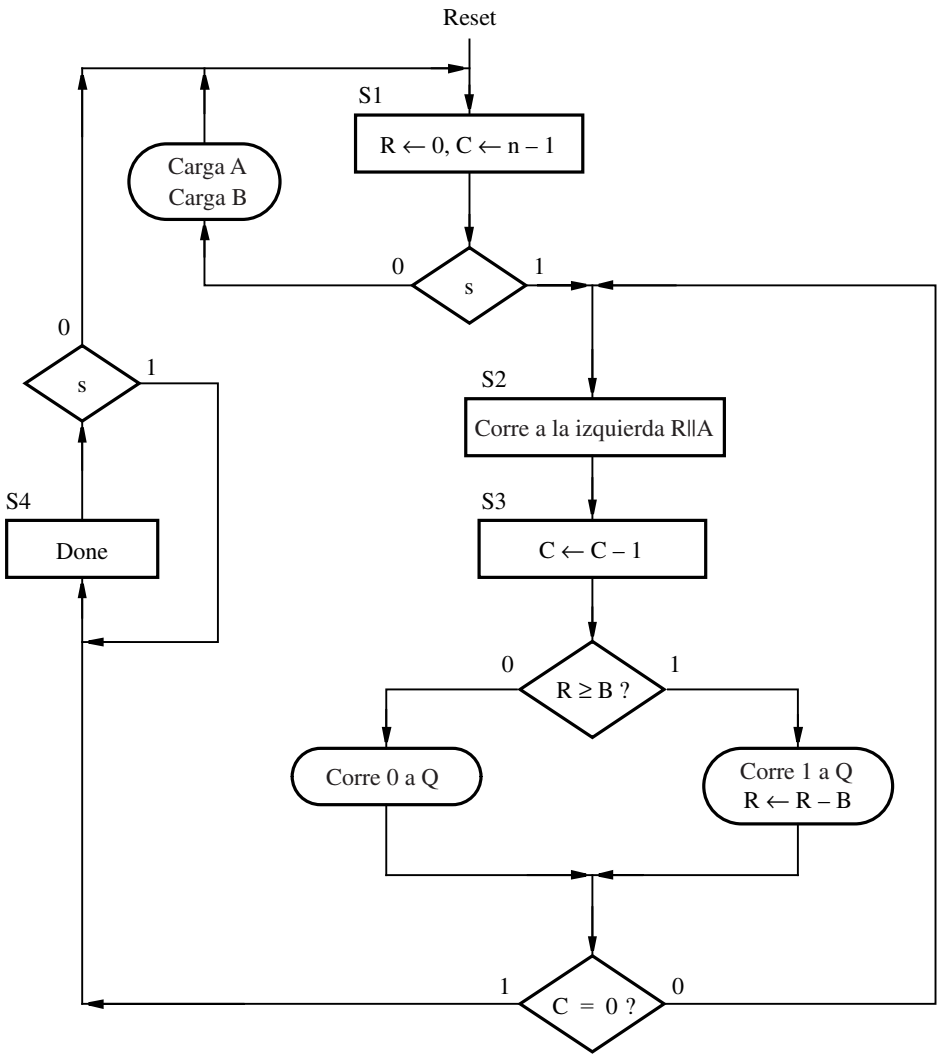


Figura 10.22 Carta ASM para el divisor.

para las entradas de datos en paralelo en *R*. El circuito de trayectoria de datos se representa en la figura 10.23. Obsérvese que el contador descendente requerido para implementar *C* y la compuerta NOR que produce como salida un 1 cuando $C = 0$ no se muestran en la figura.

Circuito de control

Una carta ASM que muestra sólo las señales de control necesarias para el divisor se presenta en la figura 10.24. En el estado *S*3 el valor de c_{out} determina si la salida de la suma del sumador se carga en *R* o no. La habilitación del corrimiento en Q se activa en el estado *S*3. No tenemos que especificar si se carga 1 o 0 en Q porque c_{out} está conectado a la entrada serial de Q en el

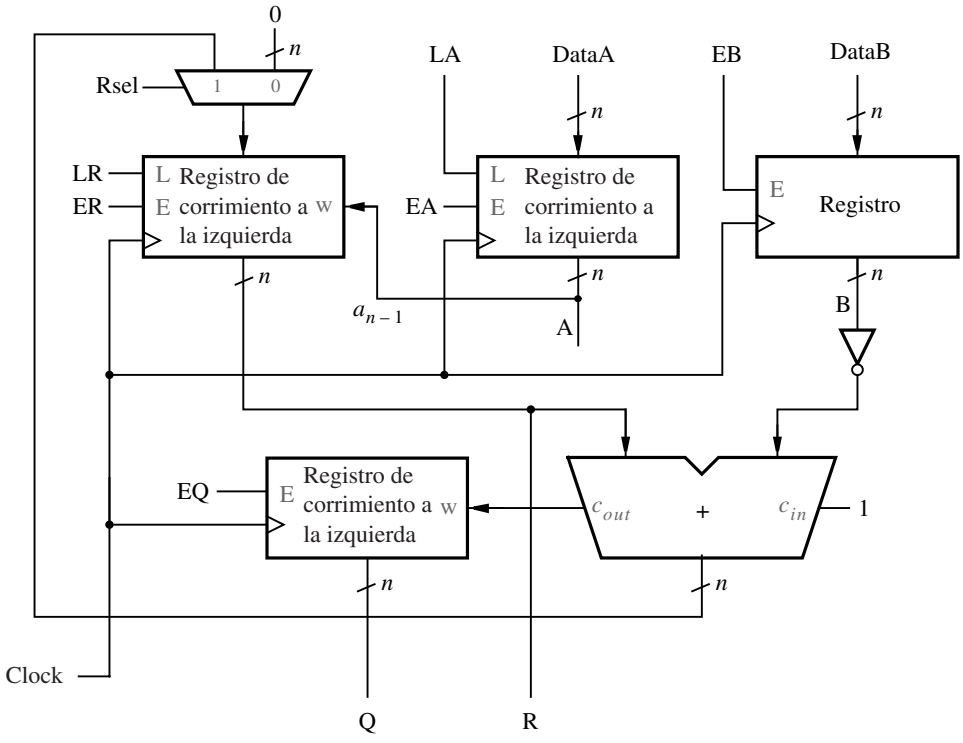


Figura 10.23 Circuito de trayectoria de datos para el divisor.

circuito de trayectoria de datos. Dejamos como ejercicio para el lector escribir el código de VHDL que representa la carta ASM de la figura 10.24 y el circuito de trayectoria de datos de la figura 10.23.

Mejoras al circuito divisor

El uso de la carta ASM de la figura 10.24 hace que el circuito cree un ciclo por los estados *S*2 y *S*3 para 2*n* ciclos de reloj. Si estos estados pueden fusionarse en uno solo, entonces el número de ciclos de reloj requeridos puede reducirse a *n*. En el estado *S*3, si $c_{out} = 1$ cargamos en *R* la salida de la suma (el resultado de la resta) desde el sumador y (suponiendo que $z = 0$) cambiamos al estado *S*2. En este estado recorremos *R* (y *A*) a la izquierda. Para combinar *S*2 y *S*3 en un estado nuevo, llamado *S*2, necesitamos poder colocar la suma en los bits del extremo izquierdo de *R* mientras al mismo tiempo se desplaza el MSB de *A* hacia el LSB de *R*. Este paso puede lograrse usando un flip-flop separado para el LSB de *R*. Llamemos rr_0 a la salida de este flip-flop. Se inicializa en 0 cuando $s = 0$ en el estado *S*1. De lo contrario, el flip-flop se carga con el MSB de *A*. En el estado *S*2, si $c_{out} = 0$, *R* se desplaza a la izquierda y rr_0 se recorre hacia *R*. Pero si $c_{out} = 1$, *R* se carga en paralelo desde las salidas de la suma del sumador.

En la figura 10.25 se ilustra cómo el ejemplo de división de la figura 10.21*b* puede realizarse usando *n* ciclos de reloj. La tabla de la figura muestra los valores de *R*, rr_0, *A*, y Q en cada paso de la división. En el circuito de trayectoria de datos de la figura 10.23 usamos un registro

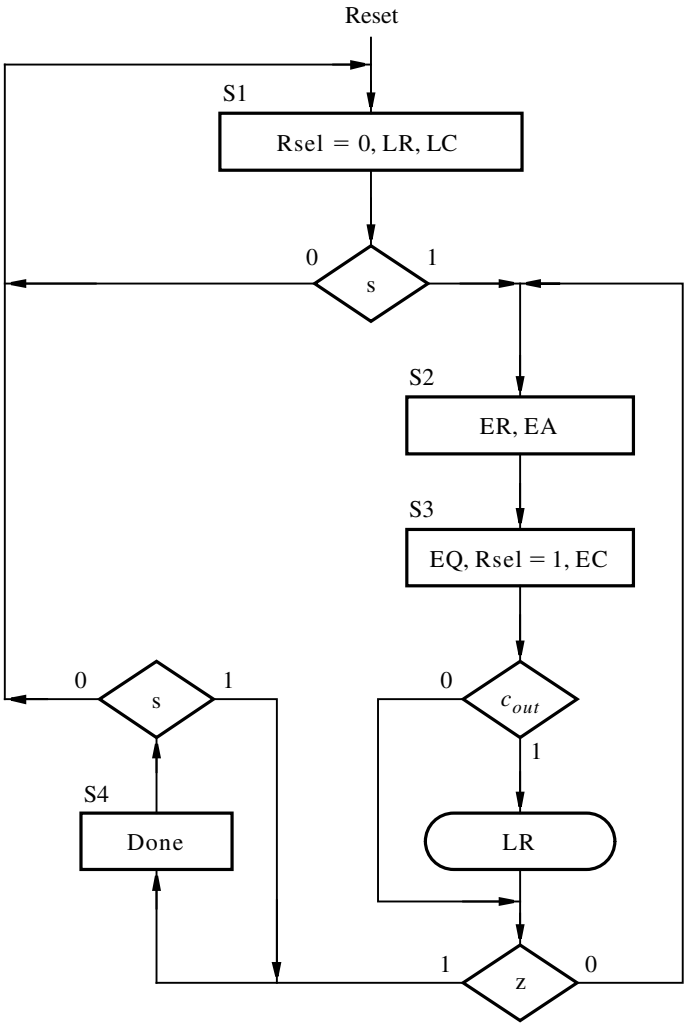


Figura 10.24 Carta ASM del circuito de control para el divisor.

de corrimiento separado para Q. Este registro en realidad no es necesario porque los dígitos del cociente pueden desplazarse hacia el bit menos significativo del registro utilizado para *A*. En la figura 10.25 los dígitos de Q que se desplazan hacia *A* se muestran en gris. La primera fila de la tabla representa la carga de datos iniciales hacia los registros *A* (y *B*) y el borrado de *R* y de rr_0 a 0. En la segunda fila de la tabla, etiquetada con el ciclo de reloj 0, la flecha gris en diagonal muestra que el bit en el extremo izquierdo de *A* (1) se recorre hacia rr_0. El número en $R||rr_0$ ahora es 000000001, que es más pequeño que *B* (1001). En el ciclo de reloj 1, rr_0 se desplaza hacia *R,* y el MSB de *A* se recorre hacia rr_0. Además, como se muestra en gris, un 0 se mueve hacia el

B → 1001 ⟩ 10001100 ← A

	Ciclo de reloj	R								rr_0	A/Q							
	Carga A, B	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0
0	Shift left	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
1	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0
2	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0
3	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
4	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
5	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1
6	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
7	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1
8	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1

Figura 10.25 Un ejemplo de división usando $n = 8$ ciclos de reloj.

LSB de Q (A). El número en $R||rr_0$ ahora es 000000010, el cual todavía es menor que B. Por consiguiente, en el ciclo de reloj 2 se realizan las mismas acciones que para el ciclo de reloj 1. Esas acciones también se llevan a cabo en los ciclos de reloj 3 y 4, en cuyo punto $R||rr_0 =$ 000010001. Como esto es mayor que B, en el ciclo de reloj 5 el resultado de la resta 000010001 $-$ 1001 = 00001000 se carga en R. El MSB de A (1) aún se desplaza hacia rr_0 y un 1 se recorre hacia Q. En los ciclos de reloj 6, 7 y 8 el número en $R||rr_0$ es mayor que B; por consiguiente, en cada uno de estos ciclos el resultado de la resta $R||rr_0 - B$ se carga en R, y un 1 se carga en Q. Después del ciclo de reloj 8 se obtiene el resultado correcto, Q = 00001111 y R = 00000101. El bit rr_0 no es parte del resultado final.

Una carta ASM que muestra los valores de las señales de control requeridas para el divisor mejorado se representa en la figura 10.26. La señal ER0 se usa junto con el flip-flop que tiene la salida rr_0. Cuando ER0 = 0, el valor 0 se carga en el flip-flop. Cuando ER0 se establece en 1 el MSB del registro de corrimiento A se carga en el flip-flop. En el estado $S1$, si $s = 0$ entonces LR se activa para inicializar R en 0. Los registros A y B pueden cargarse con datos de las entradas externas. Cuando s cambia a 1, la máquina hace una transición al estado $S2$ y simultáneamente desplaza $R||R0||A$ a la izquierda. En el estado $S2$, si $c_{out} = 1$, entonces R se carga en paralelo desde las salidas de la suma del sumador. De manera simultánea, $R0||A$ se desplaza a la izquierda (rr_0 no se recorre hacia R en este caso). Si $c_{out} = 0$, entonces $R||R0||A$ se desplaza a la izquierda. La carta ASM muestra cómo las entradas de carga en paralelo y enable en los registros deben controlarse para lograr la operación deseada.

El circuito de trayectoria de datos para el divisor mejorado se ilustra en la figura 10.27. Como vimos para la figura 10.25, los dígitos del cociente Q se desplazan hacia el registro A. Obsérvese que una de las entradas de datos de n bits en el módulo sumador está compuesta de los $n - 1$ bits menos significativos en el registro R concatenado con el bit rr_0 de la derecha.

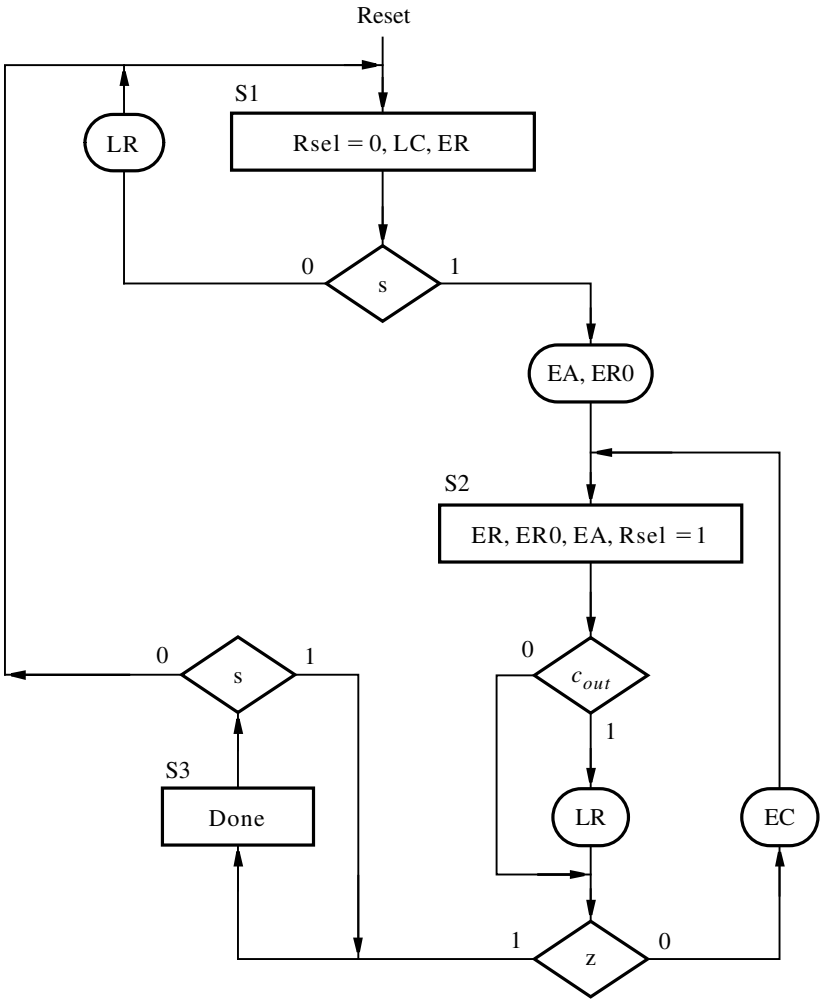


Figura 10.26 Carta ASM para el circuito de control del divisor mejorado.

Código de VHDL

En la figura 10.28 aparece el código de VHDL que representa el divisor mejorado. El parámetro genérico *N* establece el número de bits de los operandos. Los procesos *FSM_transitions* y *FSM_outputs* describen el circuito de control, igual que en los ejemplos anteriores. Al final del código se instancian los registros de corrimiento y los contadores en el circuito de trayectoria de datos. En la figura 10.28 la señal rr_0 se representa en el código mediante la señal *R*0, la cual se implementa como la salida del componente *muxdff*; el código para este subcircuito se halla en la figura 7.48. Nótese que el sumador que produce la señal *Sum* tiene una entrada definida como la concatenación de *R* con *R*0. El multiplexor necesario para la entrada a *R* se representa con la señal *DataR*. En vez de describirlo con una instrucción FOR GENERATE como en los ejemplos previos, hemos empleado la asignación de señal condicional mostrada en la parte final del código.

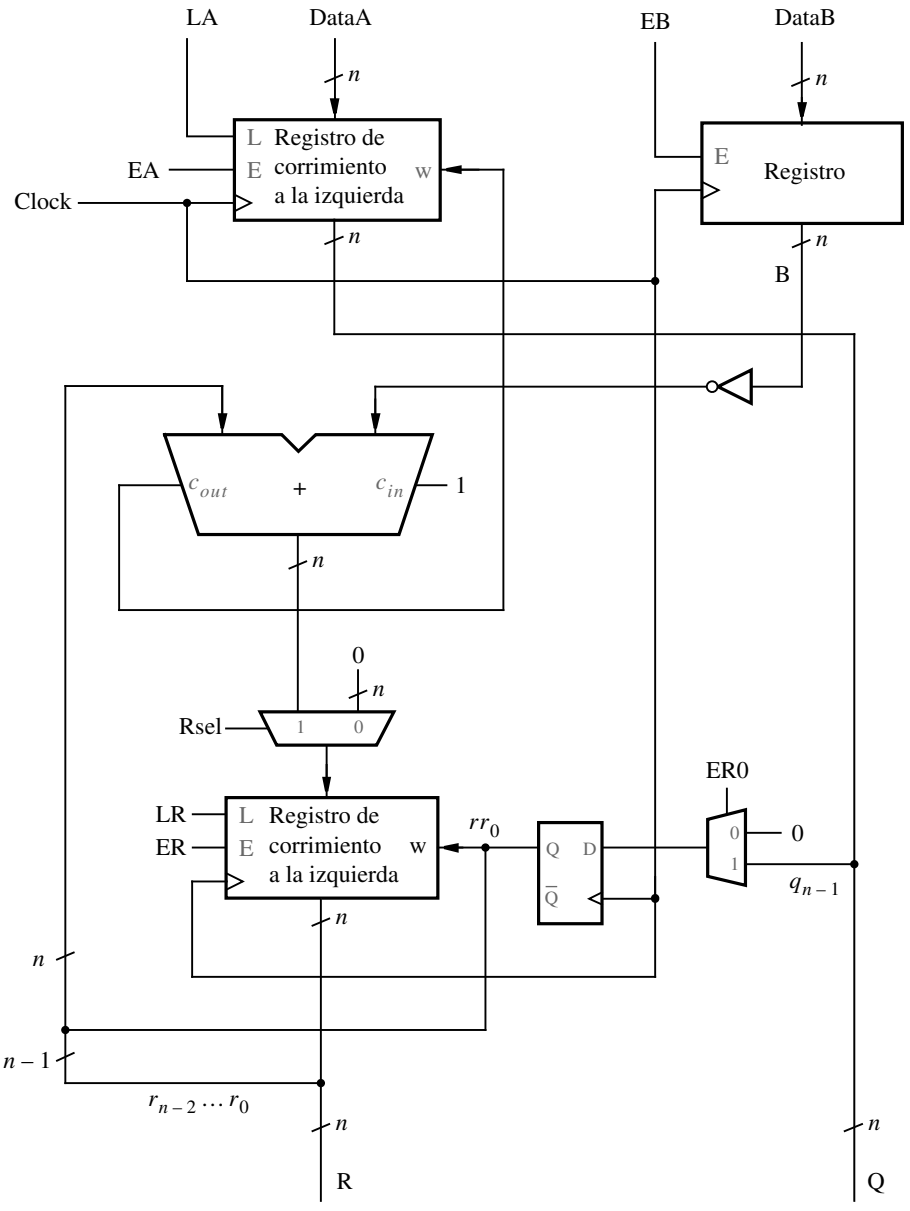


Figura 10.27 Circuito de trayectoria de datos para el divisor mejorado.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY divider IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT( Clock      : IN     STD_LOGIC ;
          Resetn     : IN     STD_LOGIC ;
          s, LA, EB  : IN     STD_LOGIC ;
          DataA      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          DataB      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          R, Q       : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          Done       : OUT    STD_LOGIC ) ;
END divider ;

ARCHITECTURE Behavior OF divider IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL Zero, Cout, z : STD_LOGIC ;
    SIGNAL EA, Rsel, LR, ER, ER0, LC, EC, R0 : STD_LOGIC ;
    SIGNAL A, B, DataR : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL Sum : STD_LOGIC_VECTOR(N DOWNTO 0) ; - - salidas del sumador
    SIGNAL Count : INTEGER RANGE 0 TO N-1 ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

. . . continúa en el inciso *b*

Figura 10.28 Código de VHDL para el circuito divisor (inciso *a*).

```
    FSM_outputs: PROCESS ( s, y, Cout, z )
    BEGIN
        LR <= '0' ; ER <= '0' ; ER0 <= '0' ;
        LC <= '0' ; EC <= '0' ; EA <= '0' ; Done <= '0' ;
        Rsel <= '0' ;
        CASE y IS
            WHEN S1 =>
                LC <= '1' ; ER <= '1' ;
                IF s = '0' THEN
                    LR <= '1' ; EA < '0' ; ER0 <= '0' ;
                ELSE
                    LR < 0 ; EA <= '1' ; ER0 <= '1' ;
                END IF ;
            WHEN S2 =>
                Rsel <= '1' ; ER <= '1' ; ER0 <= '1' ; EA <= '1' ;
                IF Cout = '1' THEN LR <= '1' ; ELSE LR <= '0' ; END IF ;
                IF z = '0' THEN EC <= '1' ; ELSE EC <= '0' ; END IF ;
            WHEN S3 =>
                Done <= '1' ;
        END CASE ;
    END PROCESS ;
    -- define el circuito de trayectoria de datos
    Zero <= '0' ;
    RegB: regne GENERIC MAP ( N => N )
        PORT MAP ( DataB, Resetn, EB, Clock, B ) ;
    ShiftR: shiftlne GENERIC MAP ( N => N )
        PORT MAP ( DataR, LR, ER, R0, Clock, R ) ;
    FF_R0: muxdff PORT MAP ( Zero, A(N-1), ER0, Clock, R0 ) ;
    ShiftA: shiftlne GENERIC MAP ( N => N )
        PORT MAP ( DataA, LA, EA, Cout, Clock, A ) ;
    Q <= A ;
    Counter: downcnt GENERIC MAP ( modulus => N )
        PORT MAP ( Clock, EC, LC, Count ) ;
    z <= '1' WHEN Count = 0 ELSE '0' ;

    Sum <= R & R0 + (NOT B + 1) ;
    Cout <= Sum(N) ;
    DataR <= (OTHERS => '0') WHEN Rsel = '0' ELSE Sum ;
END Behavior ;
```

Figura 10.28 Código de VHDL para el circuito divisor (inciso *b*).

En la figura 10.29 se observa el resultado de una simulación para el circuito producido a partir del código. Los datos $A = A6$ y $B = 8$ se cargan y luego s se establece en 1. El circuito cambia al estado $S2$ y simultáneamente desplaza R, $R0$ y A a la izquierda. La salida del registro de corrimiento que almacena a A se etiqueta Q en los resultados de la simulación porque este registro de corrimiento contiene el cociente cuando la operación de división está completa. En los primeros tres flancos activos del reloj en el estado $S2$, el número representado por $R||R0$ es menor que el número en B (8); por consiguiente, $R||R0||A$ se recorre a la izquierda en cada flanco del reloj y un 0 se desplaza hacia Q. En el cuarto ciclo consecutivo de reloj para el que la FSM ha permanecido en el estado $S2$, el contenido de R es $00000101 = (5)_{10}$, y $R0$ es 0; por tanto, $R||R0 = 000001010 = (10)_{10}$. En el siguiente flanco activo del reloj, la salida del sumador, que es $10 - 8 = 2$, se carga en R y un 1 se desplaza hacia Q. Después de n ciclos de reloj en el estado $S2$, el circuito cambia al estado $S3$ y se obtiene el resultado correcto, $Q = 14 = (20)_{10}$ y $R = 6$.

10.2.5 MEDIA ARITMÉTICA

Supóngase que k números de n bits se almacenan en una serie de registros $R_0,..., R_{k-1}$. Queremos diseñar un circuito que calcule la media M de los números en los registros. El seudocódigo para un algoritmo adecuado se muestra en la figura 10.30*a*. Cada iteración del ciclo suma el contenido de uno de los registros, indicado por R_i, a una variable *Sum*. Después que se calcula la suma, M se obtiene como *Sum/k.* Suponemos que se usa la división de enteros, de modo que un residuo R, que no se muestra en el código, también se produce.

En la figura 10.30*b* se observa una carta ASM. Mientras la entrada start, s, sea 0, los registros pueden cargarse desde entradas externas. Cuando s se vuelve 1, la máquina cambia al estado $S2$,

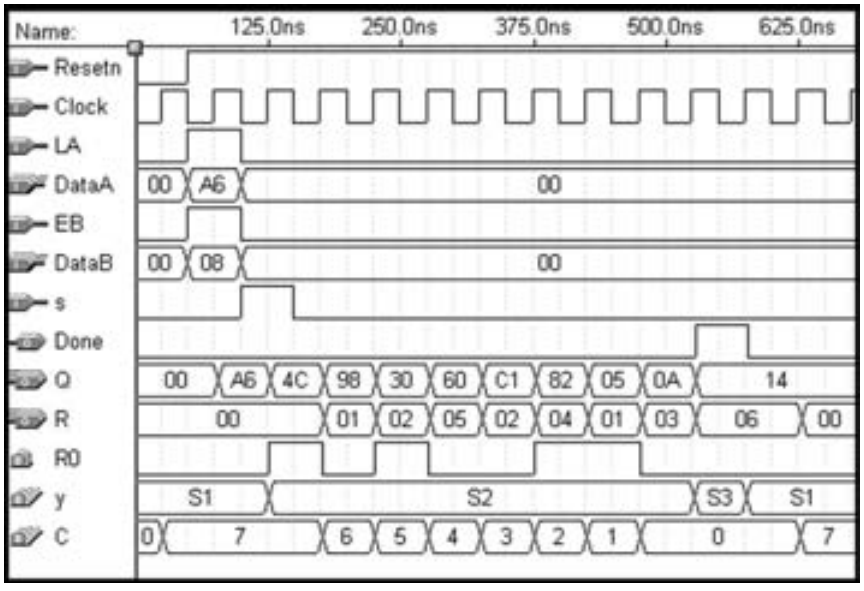
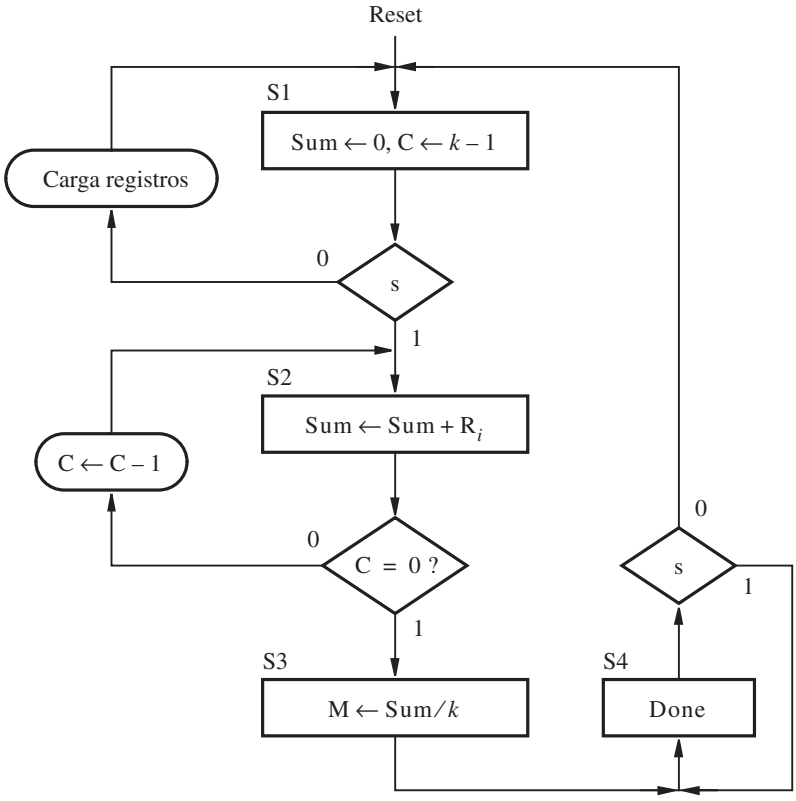


Figura 10.29 Resultados de la simulación para el circuito divisor.

```
Sum = 0 ;
for i = k − 1 down to 0 do
      Sum = Sum + R_i
end for ;
M = Sum ÷ k ;
```

a) Seudocódigo



b) Carta ASM

Figura 10.30 Un algoritmo para hallar la media de k números.

donde permanece mientras $C \neq 0$, y calcula la suma *(C* es un contador que representa *i* en la figura 10.30*a*). Cuando $C = 0$, la máquina cambia al estado *S*3 y calcula $M = Sum/k$. A partir del ejemplo anterior, sabemos que la operación de división requiere múltiples ciclos de reloj, pero hemos elegido no indicarlo en la carta ASM. Después de calcular la operación de división, se entra en el estado *S*4 y *Done* se establece en 1.

Circuito de trayectoria de datos

El circuito de trayectoria de datos para esta tarea es más complejo que nuestros ejemplos anteriores. Se representa en la figura 10.31. Necesitamos un registro con una entrada enable para almacenar *Sum.* Por razones de simplicidad, supóngase que la suma puede representarse en *n* bits sin desbordarse. Se requiere un multiplexor en las entradas de datos del registro *Sum* para seleccionar 0 en el estado *S*1 y las salidas de la suma de un sumador en el estado *S*2. El registro *Sum* proporciona una de las entradas de datos al sumador. La otra entrada debe seleccionarse de una de las salidas de datos de los *k* registros. Una forma de hacer una selección entre los registros es conectarlos a las entradas de datos de un multiplexor *k* a uno conectado al sumador. Las líneas select en el multiplexor pueden controlarse por medio del contador *C.* Para calcular la operación de división podemos usar el circuito divisor diseñado en la sección 10.2.4.

El circuito de la figura 10.31 se basa en $k = 4$, pero la misma estructura sirve para valores más grandes de *k.* Nótese que las entradas enable en los registros de R_0 a R_3 están conectadas a las salidas de un decodificador dos a cuatro que tiene la entrada de dos bits *RAdd,* la cual representa la "dirección del registro". La entrada enable del decodificador está manejada por la señal *ER*. Todos los registros se cargan desde las mismas líneas de entrada, *Data.* Como $k = 4$, podemos realizar la operación de división simplemente recorriendo *Sum* dos bits a la derecha, lo que puede hacerse en un ciclo de reloj con un registro de corrimiento que desplaza dos dígitos. Para obtener un circuito más general que funcione para cualquier valor de *k* usamos el circuito divisor diseñado en la sección 10.2.4.

Circuito de control

En la figura 10.32 se presenta un ejemplo de una carta ASM de la FSM requerida para controlar el circuito de la figura 10.31. Mientras se halla en el estado *S*1, los datos pueden cargarse en los registros $R_0,\ldots, R_{k-1}$. Pero ninguna señal de control debe activarse para este propósito, porque los registros se cargan bajo el control de las entradas *ER* y *RAdd*, como vimos antes. Cuando $s = 1$, la FSM cambia al estado *S*2, donde se activa la entrada enable *ES* en el registro *Sum* y permite que *C* disminuya. Cuando el contador llega a 0 ($z = 1$), la máquina entra en el estado *S*3, donde activa las señales *LA* y *EB* para cargar *Sum* y *k* en las entradas *A* y *B* del circuito divisor, respectivamente. La FSM entra entonces en el estado *S*4 y valida la señal *Div* para empezar la división. Cuando termina, el circuito divisor establece $zz = 1$, y la FSM pasa al estado *S*5. La media *M* aparece en las salidas Q y *R* del circuito divisor. La señal *Div* aún ha de activarse en el estado *S*5 para impedir que el circuito divisor reinicialice sus registros. Nótese que en la carta ASM de la figura 10.30*b* sólo se muestra un estado para calcular $M = Sum/k,$ pero en la figura 10.32 los estados *S*3 y *S*4 se usan para este propósito. Es posible combinar los estados *S*3 y *S*4, lo cual se dejará como un ejercicio para el lector (problema 10.6).

Circuito de trayectoria de datos alternativo

En la figura 10.31 los registros $R_0,\ldots, R_{k-1}$ están conectados al sumador por medio de un multiplexor. Otra forma de lograr la conexión deseada es añadir buffers triestado a las salidas de los *k* registros y conectar todos ellos para una posición de bit dada a la entrada correspondiente

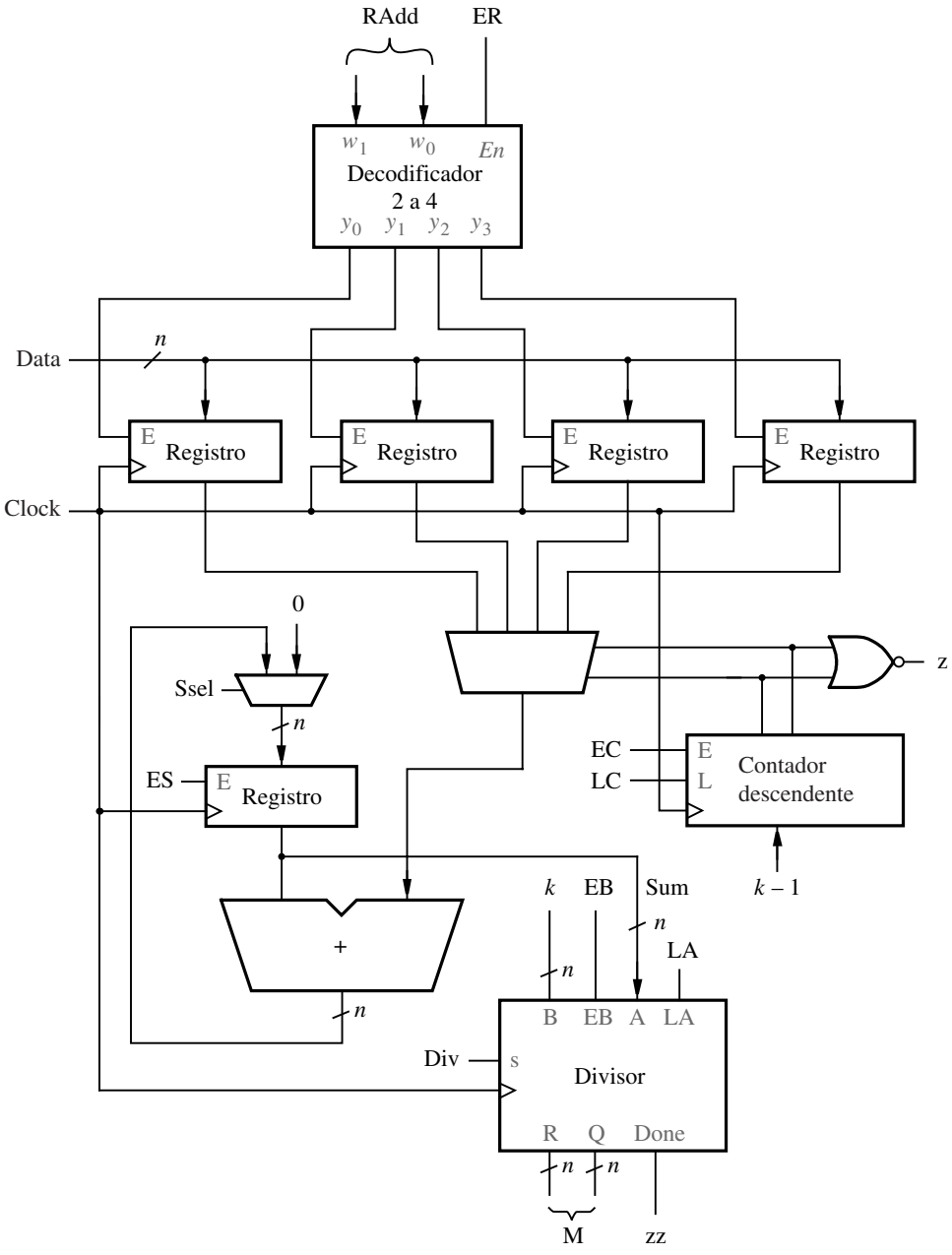


Figura 10.31 Circuito de trayectoria de datos para la operación de la media.

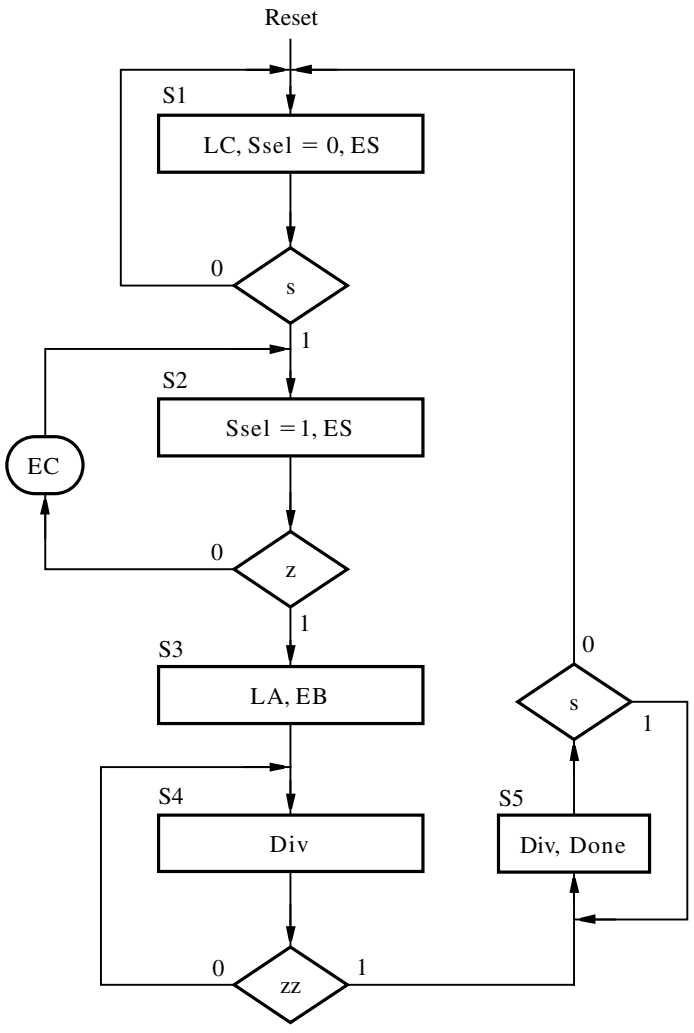


Figura 10.32 Carta ASM para el circuito de control.

en el sumador. El contador descendente *C* puede usarse para habilitar cada buffer triestado en el momento adecuado (cuando la FSM está en el estado *S*2), conectando un decodificador dos a cuatro a las salidas del contador y usando una salida del decodificador para habilitar cada uno de esos buffers. Mostraremos un ejemplo de cómo utilizar buffers triestado de esta manera en la figura 10.42.

Para valores grandes de *k* es preferible utilizar un bloque de SRAM con *k* filas y *n* columnas en vez de usar *k* registros. Los módulos predefinidos que representan los bloques de SRAM por lo general vienen incluidos en las herramientas CAD. Si el circuito que se está diseñando va a implementarse en un chip personalizado, entonces las herramientas CAD aseguran que el bloque de

SRAM buscado se incluya en el chip. Algunos PLD incluyen bloques de SRAM que pueden configurarse para implementar diversos números de filas y columnas. El sistema CAD que acompaña al libro proporciona el módulo *lpm_ram_dq*, que forma parte de la biblioteca estándar LPM.

En la figura 10.33 se observa un diagrama esquemático para el circuito de la media aritmética, donde se emplean los parámetros $k = 16$ y $n = 8$. Ese esquema se creó con las herramientas CAD que acompañan al libro. Cuatro de los símbolos gráficos del esquema representan subcircuitos descritos con código de VHDL: *downcnt, regne, divider* y *meancntl.* El código para el subcircuito *divider* se muestra en la figura 10.28. El subcircuito *meancntl* representa la FSM de la figura 10.32. El código de VHDL para esta FSM no se muestra. El esquema también incluye un multiplexor conectado al registro *Sum*, un sumador y una compuerta NOR que detecta cuándo el contador *C* llega a 0. Las salidas del contador proporcionan las entradas de dirección al bloque de SRAM, llamado *MReg.*

El bloque de SRAM tiene 16 filas y ocho columnas. En la figura 10.31 un decodificador controla la carga de datos en cada uno de los *k* registros. Para leer los datos desde los registros se usa el contador *C.* A fin de mantener simple el esquema de la figura 10.33, hemos incluido el

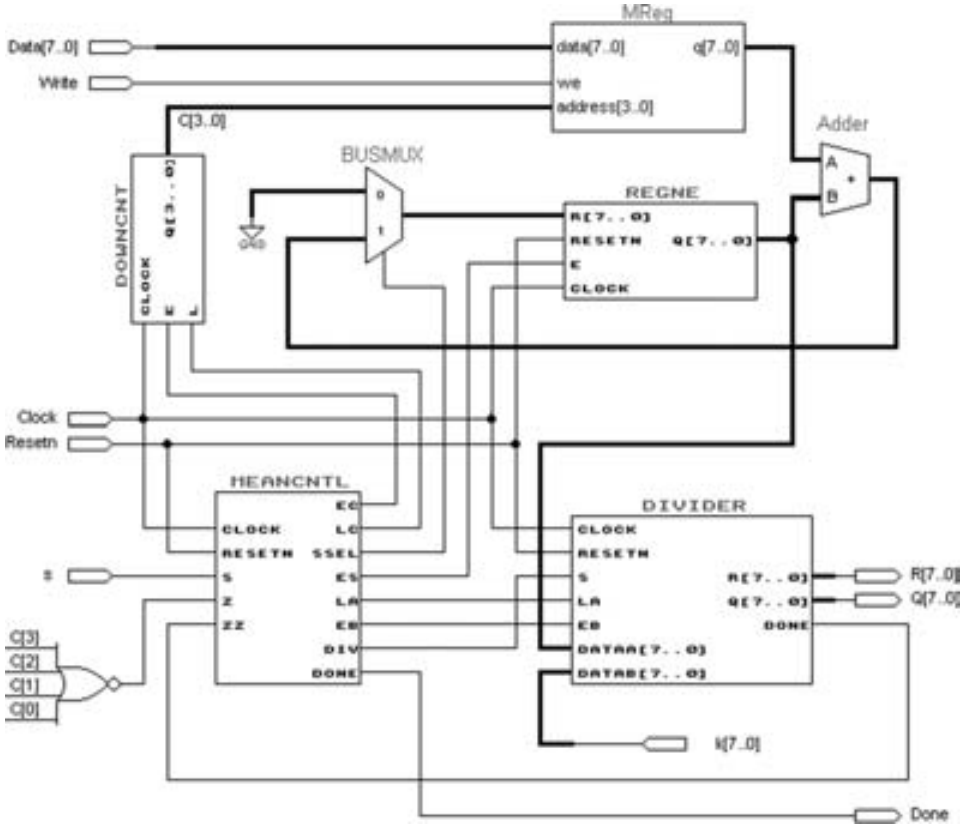


Figura 10.33 Esquema del circuito de la media con un bloque de SRAM.

contador para leer datos desde el bloque de SRAM, pero hemos ignorado la cuestión de escribirlos en él. Es posible modificar el código de *meancntl* para permitir que el contador *C* se dirija al bloque de SRAM a fin de cargar los datos iniciales, pero no ahondaremos en este aspecto aquí.

Con fines de simulación podemos usar una función del sistema CAD que permite que los datos iniciales se almacenen en el bloque de SRAM. Elegimos almacenar 0 en R_0 (fila 0 del bloque de SRAM); 1 en R_1,…, y 15 en R_{15}. Los resultados de una simulación de tiempo para el circuito implementado en un chip FPGA se muestran en la figura 10.34. Sólo una parte de la simulación, desde el punto donde $C = 5$, se muestra en la figura. En ese punto la FSM *meancntl* se halla en el estado *S*2 y la suma *Sum* se está acumulando. Cuando *C* llega a 0 *Sum* tiene el valor correcto, que es $0 + 1 + 2 + \ldots + 15 = 120 = (78)_{16}$. La FSM cambia al estado *S*3 durante un ciclo de reloj y luego permanece en el estado *S*4 hasta que la operación de división se termina. El resultado correcto, Q = 7 y $R = 8$, se obtiene cuando la FSM cambia al estado *S*5.

10.2.6 OPERACIÓN DE ORDENACIÓN

Dada una lista de *k* números de *n* bits sin signo almacenados en una serie de registros $R_0,\ldots, R_{k-1}$, queremos diseñar un circuito que pueda ordenar el contenido de los registros en orden ascendente. El seudocódigo para un algoritmo de ordenación simple se muestra en la figura 10.35. Se basa en hallar el número menor en la sublista $R_i,\ldots, R_{k-1}$ y mover ese número a R_i, para $i = 1, 2,\ldots, k-2$. Cada iteración del ciclo externo coloca el número de R_i en *A*. Cada iteración del ciclo interno compara ese número con el contenido de otro registro R_j. Si el número en R_j es menor que *A,* el contenido de R_i y R_j se intercambian y *A* cambia para almacenar el contenido nuevo de R_i.

Una carta ASM que representa el algoritmo de ordenación se muestra en la figura 10.36. En el estado inicial *S*1, mientras $s = 0$ los registros se cargan desde las entradas de datos externas y un contador C_i, que representa *i* en el ciclo externo, se borra. Cuando la máquina cambia al estado *S*2*, A* se carga con el contenido de R_i. Además, C_j, que representa a *j* en el ciclo interno,

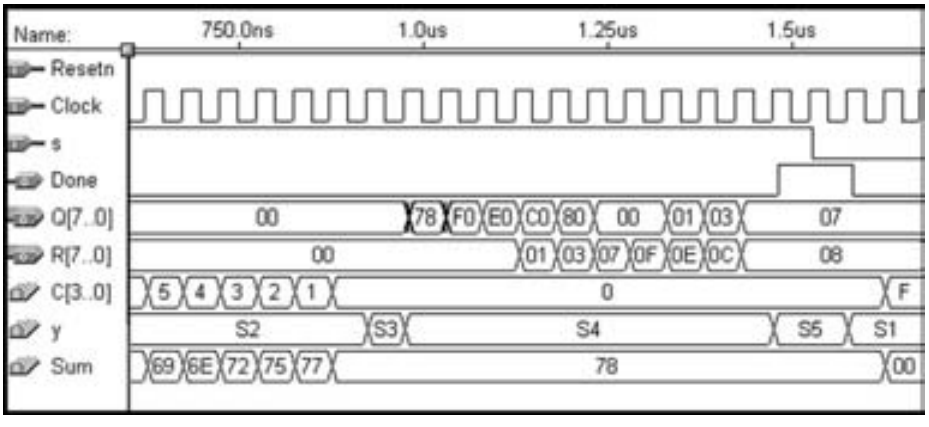


Figura 10.34 Resultados de la simulación para el circuito de la media usando SRAM.

```
for i = 0 to k − 2 do
    A = R_i ;
    for j = i + 1 to k − 1 do
        B = R_j ;
        if B < A then
            R_i = B ;
            R_j = A ;
            A = R_i ;
        end if ;
    end for ;
end for ;
```

Figura 10.35 Seudocódigo para la operación de ordenación.

se inicializa al valor de *i*. El estado *S*3 se usa para inicializar *j* al valor $i + 1$, y el estado *S*4 carga el valor de R_j en *B.* En el estado *S*5, *A* y *B* se comparan, y si $B < A$, la máquina pasa al estado *S*6. Los estados *S*6 y *S*7 intercambian los valores de R_i y R_j. El estado *S*8 carga *A* con R_i. Aun cuando este paso es necesario sólo para el caso donde $B < A$, el flujo de control es más simple si esta operación se realiza en ambos casos. Si C_j no es igual a $k - 1$ la máquina cambia de *S*8 a *S*4, y por tanto permanece en el ciclo interno. Si $C_j = k - 1$ y C_i no es igual a $k - 2$, entonces la máquina se queda en el ciclo externo al cambiar al estado *S*2.

Circuito de trayectoria de datos

Hay muchas maneras de implementar un circuito de trayectoria de datos que satisfaga los requisitos de la carta ASM de la figura 10.36. Una posibilidad se ilustra en las figuras 10.37 y 10.38. En la primera se muestra que es posible conectar los registros $R_0,\ldots, R_{k-1}$ a los registros *A* y *B* mediante multiplexores cuatro a uno. Supóngase el valor $k = 4$ por simplicidad. Los registros *A* y *B* están conectados a un subcircuito comparador y, por medio de los multiplexores, de nuevo a las entradas de los registros $R_0,\ldots, R_{k-1}$. Los registros pueden cargarse con los datos iniciales (sin ordenar) usando las líneas *DataIn*. Los datos se escriben (cargan) en cada registro al validar la señal de control *WrInit* y colocar la dirección del registro en la entrada *RAdd*. El buffer triestado manejado por la señal de control *Rd* sirve para dar salida al contenido de los registros en la salida *DataOut*.

Las señales $Rin_0,\ldots, Rin_{k-1}$ están controladas por el decodificador dos a cuatro mostrado en la figura 10.38. Si $Int = 1$, uno de los contadores C_i o C_j maneja al decodificador. Si $Int = 0$, entonces el decodificador es manejado por la entrada externa *RAdd.* Las señales z_i y z_j se establecen en 1 cuando $C_i = k - 2$ y $C_j = k - 1$, respectivamente. Una carta ASM que muestra las señales de control utilizadas en el circuito de trayectoria de datos se proporciona en la figura 10.39.

Código de VHDL

El código de VHDL para la operación de ordenación se presenta en la figura 10.40. En vez de definir señales separadas llamadas $R_0,\ldots, R_3$ para las salidas del registro hemos elegido especificar los registros como un arreglo. Este enfoque permite hacer referencia a ellos como *R(i)* en una instrucción FOR GENERATE que instancia cada registro. El arreglo de registros se define

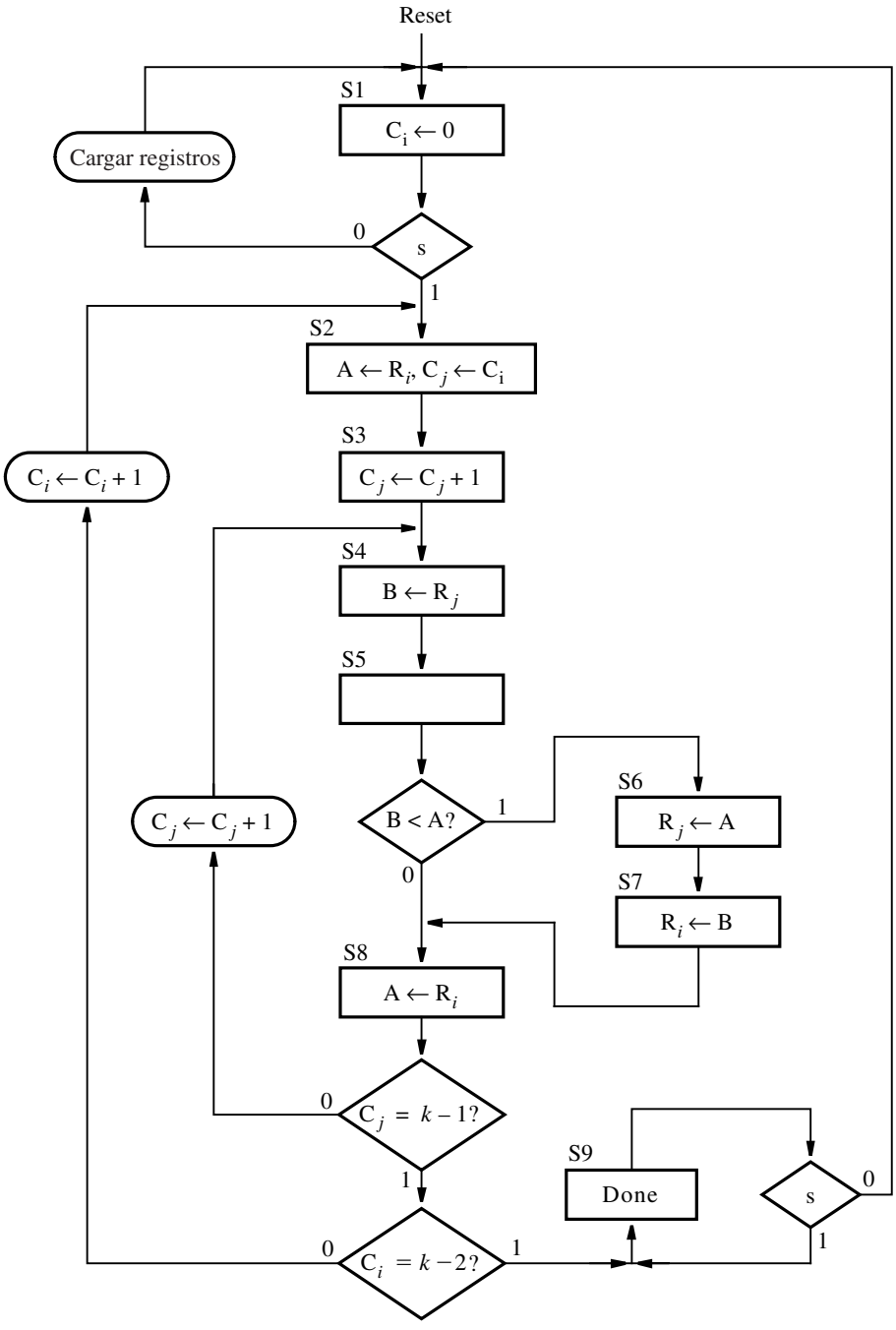


Figura 10.36 Carta ASM para la operación de ordenación.

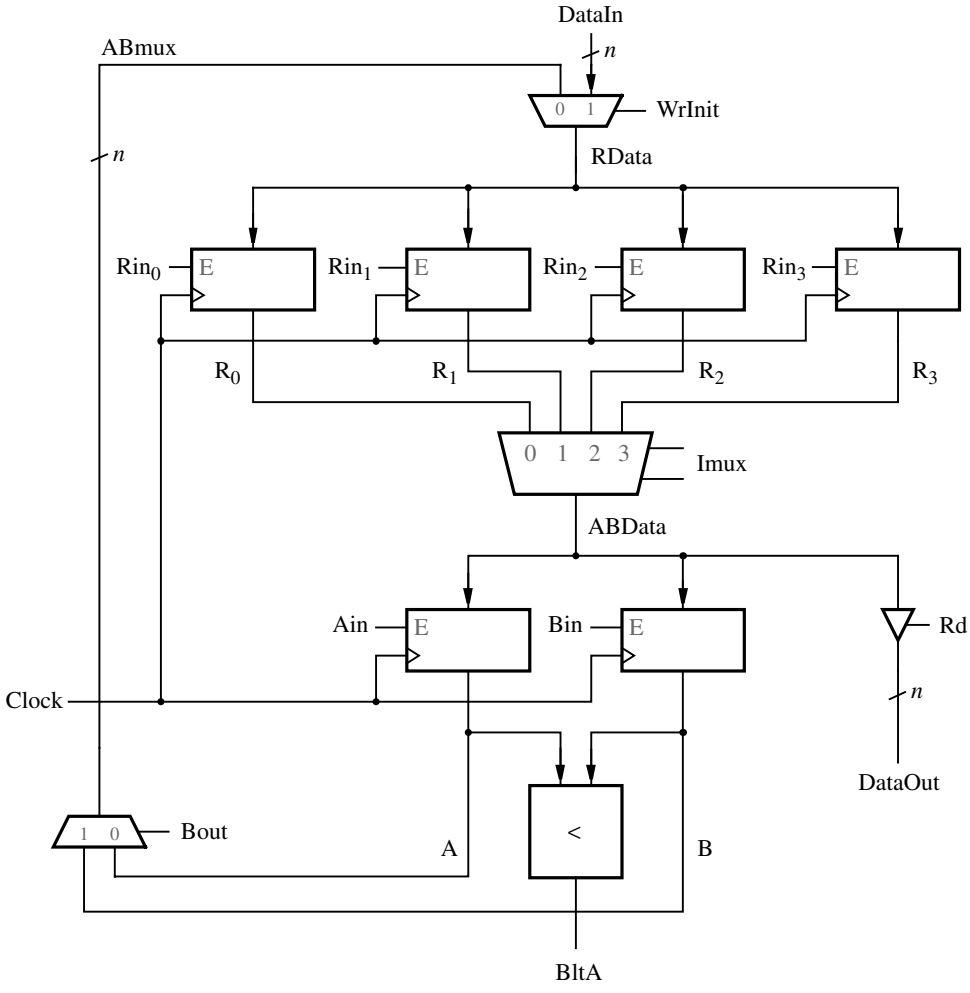


Figura 10.37 Una parte del circuito de trayectoria de datos para la operación de ordenación.

en dos pasos. Primero, un tipo definido por el usuario, para el que hemos elegido el nombre *RegArray,* se define en la instrucción

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(N-1 DOWNTO 0)
```

Esta instrucción especifica que el tipo *RegArray* representa un arreglo de cuatro señales STD_LOGIC_VECTOR. El tipo STD_LOGIC_VECTOR también se define como un arreglo en la norma del IEEE; se trata de un arreglo de señales STD_LOGIC. La señal *R* se define como un arreglo con cuatro elementos del tipo *RegArray*.

La FSM que controla la operación de ordenación se describe de la misma forma que en ejemplos anteriores, usando los procesos *FSM_transitions* y *FSM_outputs.* Después de estos

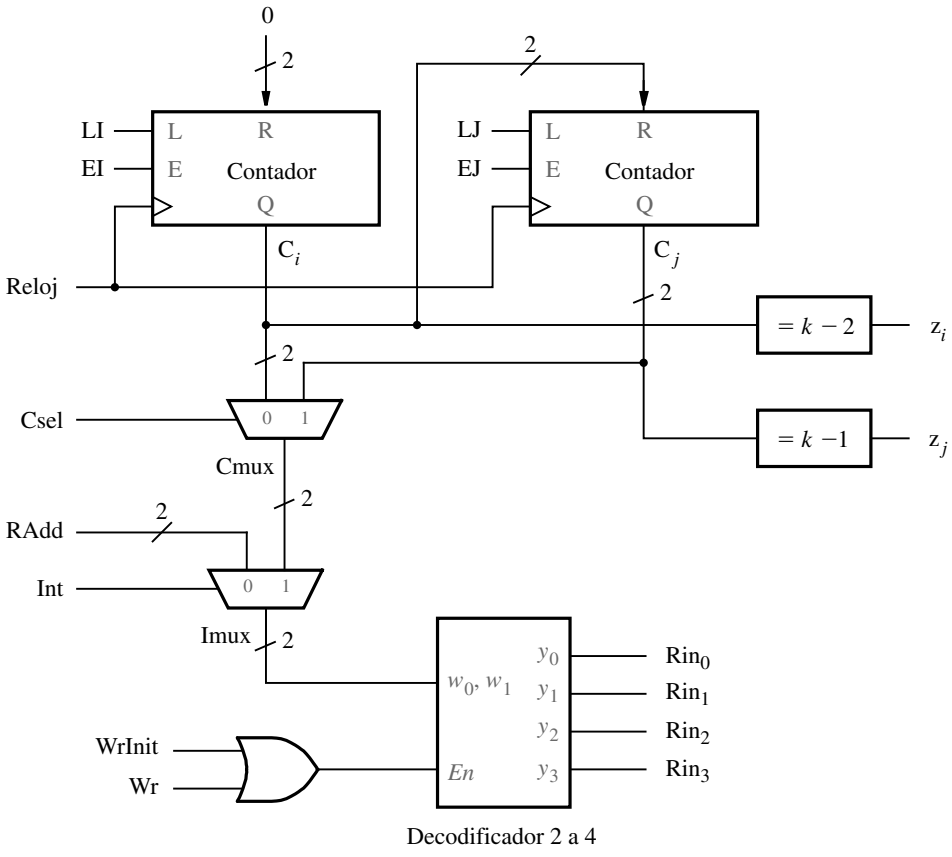


Figura 10.38 Una parte del circuito de trayectoria de datos para la operación de ordenación.

procesos, el código instancia los registros de R_0 a R_3, así como *A* y *B.* Los contadores C_i y C_j están instanciados por las dos instrucciones etiquetadas *OuterLoop* e *InnerLoop,* respectivamente. Los multiplexores con las salidas *CMux* e *IMux* se especifican usando asignaciones de señales condicionales. El multiplexor cuatro a uno de la figura 10.37 se define por medio de la asignación de señal seleccionada que especifica el valor de la señal *ABData* para cada valor de *IMux.* El decodificador dos a cuatro de la figura 10.38 con las salidas Rin_0,…, Rin_3 está definido por la instrucción de proceso llamada *RinDec.* Finalmente, las señales z_i y z_j y la salida *DataOut* se especifican utilizando asignaciones de señales condicionales.

Implementamos el código de la figura 10.40 en un chip FPGA. En la figura 10.41 se ofrece un ejemplo del resultado de una simulación. En el inciso *(a)* se muestra la primera mitad de la simulación, desde 0 hasta 1.25 μs, y en el *(b)* la segunda mitad, desde 1.25 μs hasta 2.5 μs. Después de reinicializar el circuito, *WrInit* se establece en 1 durante cuatro ciclos de reloj y los datos sin ordenar se escriben en los cuatro registros utilizando las entradas *DataIn* y *RAdd*. Después de que *s* cambia a 1, la FSM cambia al estado *S*2. Los estados *S*2 a *S*4 cargan *A* con el

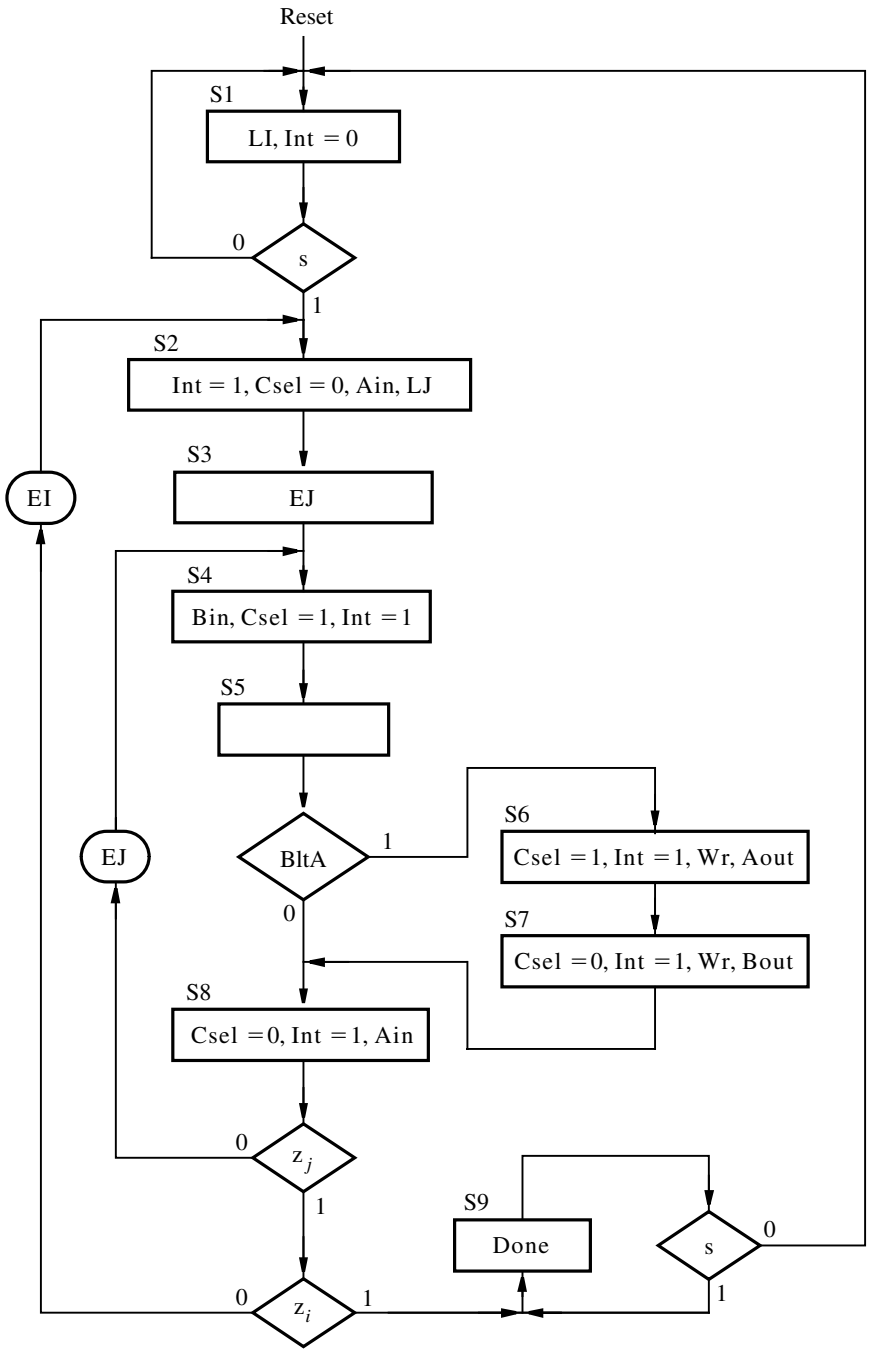


Figura 10.39 Carta ASM para el circuito de control.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.components.all ;

ENTITY sort IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( Clock, Resetn  : IN      STD_LOGIC ;
           s, WrInit, Rd  : IN      STD_LOGIC ;
           DataIn         : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           RAdd           : IN      INTEGER RANGE 0 TO 3 ;
           DataOut        : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Done           : BUFFER  STD_LOGIC ) ;
END sort ;

ARCHITECTURE Behavior OF sort IS
    TYPE State_type IS ( S1, S2, S3, S4, S5, S6, S7, S8, S9 ) ;
    SIGNAL y : State_type ;
    SIGNAL Ci, Cj : INTEGER RANGE 0 TO 3 ;
    SIGNAL Rin : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    TYPE RegArray IS
        ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL R : RegArray ;
    SIGNAL RData, ABMux : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL Int, Csel, Wr, BltA : STD_LOGIC ;
    SIGNAL CMux, IMux : INTEGER RANGE 0 TO 3 ;
    SIGNAL Ain, Bin, Aout, Bout : STD_LOGIC ;
    SIGNAL LI, LJ, EI, EJ, zi, zj : STD_LOGIC ;
    SIGNAL Zero : INTEGER RANGE 3 DOWNTO 0 ; - - datos paralelos para Ci = 0
    SIGNAL A, B, ABData : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 => IF S = '0' THEN y <= S1 ;
                    ELSE y <= S2 ; END IF ;
                WHEN S2 => y <= S3 ;
                WHEN S3 => y <= S4 ;
                WHEN S4 => y <= S5 ;
```

. . . continúa en el inciso *b*

Figura 10.40 Código de VHDL para la operación de ordenación (inciso *a*).

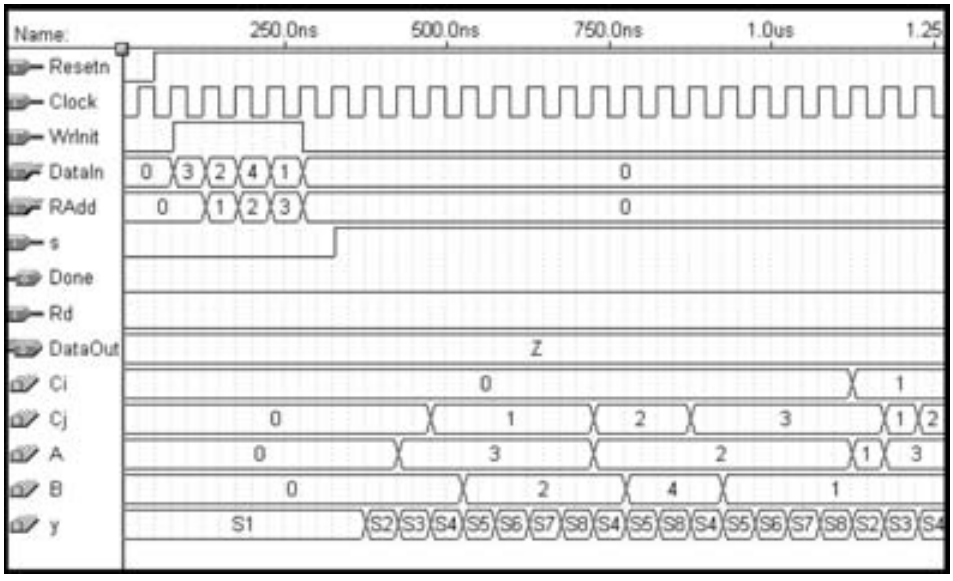
```
                WHEN S5 => IF BltA = '1' THEN y <= S6 ;
                    ELSE y <= S8 ; END IF ;
                WHEN S6 => y <= S7 ;
                WHEN S7 => y <= S8 ;
                WHEN S8 =>
                    IF zj = '0' THEN y <= S4 ;
                    ELSIF zi = '0' THEN y <= S2 ;
                    ELSE y <= S9 ;
                    END IF ;
                WHEN S9 => IF s = '1' THEN y <= S9 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    -- define las salidas generadas por la FSM
    Int <= '0' WHEN y = S1 ELSE '1' ;
    Done <= '1' WHEN y = S9 ELSE '0' ;
    FSM_outputs: PROCESS ( y, zi, zj )
    BEGIN
        LI <= '0' ; LJ <= '0' ; EI <= '0' ; EJ <= '0' ; Csel <= '0' ;
        Wr <= '0'; Ain <= '0' ; Bin <= '0' ; Aout <= '0' ; Bout <= '0' ;
        CASE y IS
            WHEN S1 => LI <= '1' ;
            WHEN S2 => Ain <= '1' ; LJ <= '1' ;
            WHEN S3 => EJ <= '1' ;
            WHEN S4 => Bin <= '1' ; Csel <= '1' ;
            WHEN S5 => -- no se activan salidas en este estado
            WHEN S6 => Csel <= '1' ; Wr <= '1' ; Aout <= '1' ;
            WHEN S7 => Wr <= '1' ; Bout <= '1' ;
            WHEN S8 => Ain <= '1' ;
                IF zj = '0' THEN
                    EJ <= '1' ;
                ELSE
                    EJ <= '0' ;
                    IF zi = '0' THEN
                        EI <= '1' ;
                    ELSE
                        EI <= '0' ;
                    END IF;
                END IF ;
            WHEN S9 => -- se asigna 1 a Done por medio de la asignación de señal condicional
        END CASE ;
    END PROCESS ;
```

. . . continúa en el inciso *c*

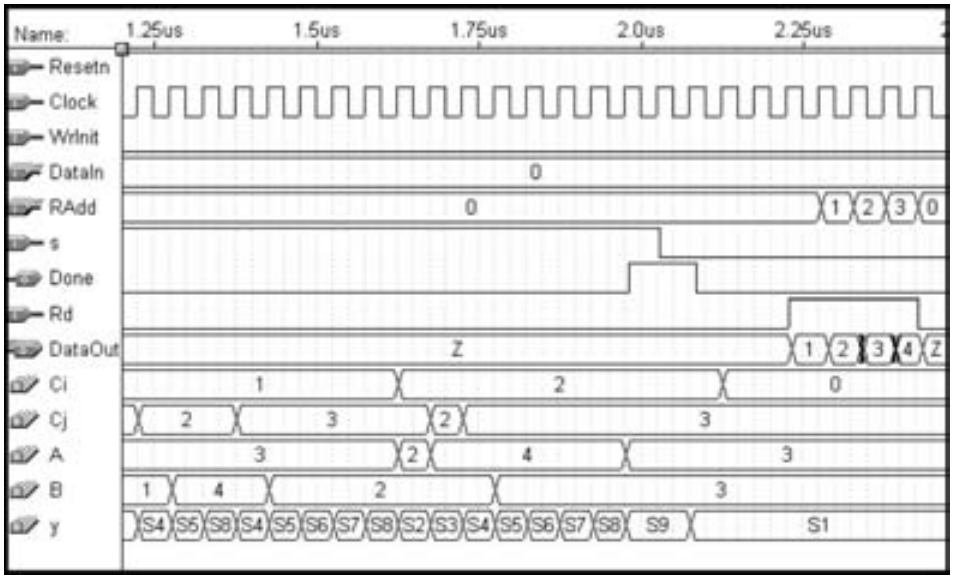
Figura 10.40 Código de VHDL para la operación de ordenación (inciso *b*).

```
    - - define el circuito de trayectoria de datos
    Zero <= 0 ;
    GenReg: FOR i IN 0 TO 3 GENERATE
        Reg: regne GENERIC MAP ( N => N )
            PORT MAP ( RData, Resetn, Rin(i), Clock, R(i) ) ;
    END GENERATE ;
    RegA: regne GENERIC MAP ( N => N )
        PORT MAP ( ABData, Resetn, Ain, Clock, A ) ;
    RegB: regne GENERIC MAP ( N => N )
        PORT MAP ( ABData, Resetn, Bin, Clock, B ) ;
    BltA <= '1' WHEN B < A ELSE '0' ;
    ABMux <= A WHEN Bout = '0' ELSE B ;
    RData <= ABMux WHEN WrInit = '0' ELSE DataIn ;
    OuterLoop: upcount GENERIC MAP ( modulus => 4 )
        PORT MAP ( Resetn, Clock, EI, LI, Zero, Ci ) ;
    InnerLoop: upcount GENERIC MAP ( modulus => 4 )
        PORT MAP ( Resetn, Clock, EJ, LJ, Ci, Cj ) ;
    CMux <= Ci WHEN Csel = '0' ELSE Cj ;
    IMux <= Cmux WHEN Int = '1' ELSE Radd ;
    WITH IMux Select
        ABData <= R(0) WHEN 0,
                  R(1) WHEN 1,
                  R(2) WHEN 2,
                  R(3) WHEN OTHERS ;
    RinDec: PROCESS ( WrInit, Wr, IMux )
    BEGIN
        IF (WrInit OR Wr) = '1' THEN
            CASE IMux IS
                WHEN 0 => Rin <= "0001" ;
                WHEN 1 => Rin <= "0010" ;
                WHEN 2 => Rin <= "0100" ;
                WHEN OTHERS => Rin <= "1000" ;
            END CASE ;
        ELSE Rin <= "0000" ;
        END IF ;
    END PROCESS ;
    Zi <= '1' WHEN Ci = 2 ELSE '0' ;
    Zj <= '1' WHEN Cj = 3 ELSE '0' ;
    DataOut <= (OTHERS => 'Z') WHEN Rd = '0' ELSE ABData ;
END Behavior ;
```

Figura 10.40 Código de VHDL para la operación de ordenación (inciso *c*).



a) Carga de los registros y comienzo de la operación de ordenación



b) Terminación de la operación de ordenación y lectura de los registros

Figura 10.41 Resultados de la simulación para la operación de ordenación.

contenido de R_0 (3) y B con el contenido de R_1 (2). El estado $S5$ compara B con A, y como $B < A$ la FSM utiliza los estados $S6$ y $S7$ para intercambiar el contenido de los registros R_0 y R_1. En el estado $S8$, A se vuelve a cargar con R_0, que ahora contiene 2. Como z_j no se activa, la FSM incrementa el contador C_j y cambia de nuevo al estado $S4$. El registro B ahora se carga con el contenido de R_2 (4), y la FSM cambia al estado $S5$. Puesto que $B = 4$ no es menor que $A = 2$, la máquina cambia a $S8$ y luego de nuevo a $S4$. El registro B ahora se carga con el contenido de R_3 (1), que luego se compara con $A = 2$ en el estado $S5$. El contenido de R_0 y R_3 se intercambia y la máquina cambia a $S8$. En este punto, el contenido de los registros es $R_0 = 1$, $R_1 = 3$, $R_2 = 4$ y $R_3 = 2$. Puesto que $z_j = 1$ y $z_i = 0$, la FSM realiza la siguiente iteración del ciclo externo al cambiar al estado $S2$. Al avanzar en el tiempo de simulación, en la figura 10.41*b* el circuito alcanza el estado en el que $C_i = 2$, $C_j = 3$ y la FSM está en el estado $S8$. La FSM luego cambia al estado $S9$ y establece *Done* en el valor 1. Los datos ordenados correctamente son leídos desde los registros al establecer la señal $Rd = 1$ y utilizar las entradas *RAdd* para seleccionar cada uno de los registros.

Circuito de trayectoria de datos alternativo

En la figura 10.37 usamos los multiplexores para conectar los diversos registros en el circuito de trayectoria de datos. Otro método consiste en ocupar buffers triestado para interconectar los registros, como se ilustra en la figura 10.42. Como dijimos en la sección 7.14, el conjunto de n cables comunes que conecta los registros se llama *bus.* El circuito de la figura 10.42 tiene dos buses, uno que conecta las salidas de los registros $R_0,\ldots, R_3$ a las entradas de los registros A y B, y otro que conecta las salidas de A y B de nuevo a las entradas de $R_0,\ldots, R_{k-1}$. Cuando los multiplexores proporcionan la conexión entre registros, como se muestra en la figura 10.37, el término *bus* aún puede utilizarse para referirse a la conexión entre registros.

El circuito de la figura 10.42 utiliza el circuito de la figura 10.38 con una modificación. En la figura 10.38 la señal *IMux* está conectada al decodificador dos a cuatro que genera $Rin_0,\ldots, Rin_3$. Si se usa el circuito de la figura 10.42, entonces se precisa un segundo decodificador conectado a *IMux* para generar las señales de control $Rout_0,\ldots, Rout_3$. El circuito de control descrito en la carta ASM de la figura 10.39 puede utilizarse para el circuito de trayectoria de datos de la figura 10.42.

En la sección 10.2.5 dijimos que para los valores grandes de k es mejor utilizar un bloque de SRAM para almacenar los datos que usar registros individuales. El circuito de ordenación puede modificarse para que utilice un bloque de SRAM con k filas y n columnas. En este caso el circuito de trayectoria de datos es parecido al de la figura 10.37, pero no requieren los multiplexores cuatro a uno porque las salidas de datos desde el bloque de SRAM están conectadas directamente a los registros A y B. Sigue siendo necesario emplear el circuito de la figura 10.38, excepto en que no se requiere el decodificador dos a cuatro porque la señal *IMux* está conectada a las entradas de dirección en el bloque de SRAM. La entrada *write* en el bloque de SRAM está manejada por la compuerta OR con las entradas *WrInit* y *Wr.* Puede escribirse código de VHDL para el circuito de ordenación, en el cual un componente que representa el bloque de SRAM se instancia desde una biblioteca de módulos predefinidos, o se proporciona código de VHDL tal que una herramienta CAD pueda inferir la necesidad de un bloque de memoria. El código para el circuito de control mostrado en la figura 10.40 no tiene que cambiar (véase el problema 10.11).

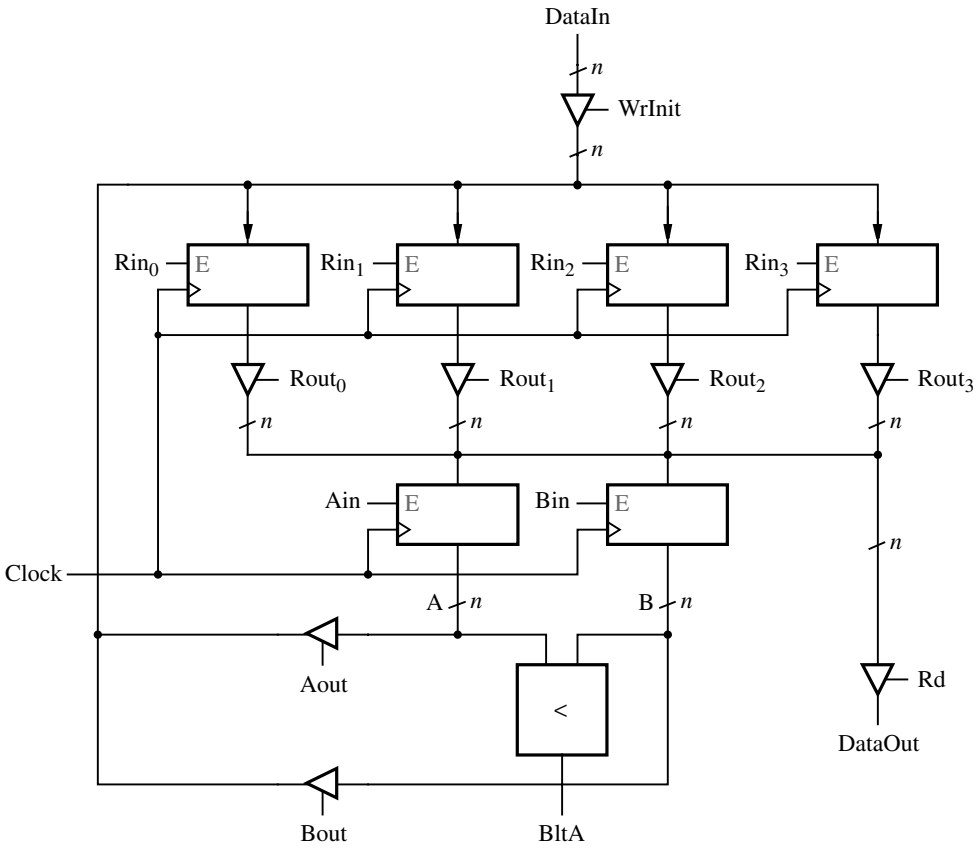


Figura 10.42 Uso de los buffers triestado en el circuito de trayectoria de datos.

10.3 SINCRONIZACIÓN DEL RELOJ

En la sección anterior brindamos varios ejemplos de circuitos que contienen muchos flip-flops. En el capítulo anterior mostramos que para asegurar la operación adecuada de los circuitos secuenciales es indispensable considerar detenidamente los aspectos de la sincronización asociados con los elementos de almacenamiento. En esta sección abordaremos algunas cuestiones relativas a la sincronización de los circuitos secuenciales.

10.3.1 DESVIACIÓN DEL RELOJ

En la figura 10.1 se muestra cómo una entrada enable puede usarse para evitar que un flip-flop cambie su valor almacenado cuando ocurre un flanco activo del reloj. Otra forma de implementar la función enable del reloj se muestra en la figura 10.43. El circuito utiliza una compuerta AND para forzar a que la entrada del reloj tenga el valor de 0 cuando $E = 0$. Este circuito es más simple que el de la figura 10.1, pero puede ocasionar problemas en la práctica. Considérese un circuito secuencial que tiene muchos flip-flops, algunos de los cuales tienen una entrada enable

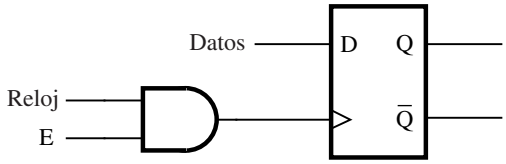


Figura 10.43 Circuito para habilitar el reloj.

y otros no. Si se utiliza el circuito de la figura 10.43, entonces los flip-flops sin la entrada enable observarán cambios en la señal de reloj ligeramente antes que los flip-flops con la entrada enable. Esta situación, en la que la señal de reloj llega en tiempos diferentes a distintos flip-flops, se conoce como *desviación del reloj.* En la figura 10.43 se muestra sólo una fuente posible de desviación del reloj. Problemas parecidos surgen en un chip en el que la señal de reloj se distribuye a diferentes flip-flops mediante cables cuyas longitudes varían notoriamente.

Para comprender los problemas que posiblemente ocasione la desviación del reloj, considérese el circuito de trayectoria de datos para el ejemplo de conteo de bits de la figura 10.11. El LSB del registro de corrimiento, a_0, se utiliza como una señal de control que determina si un contador se incrementa o no. Supóngase que existe una desviación del reloj que causa que la señal de reloj llegue antes a los flip-flops del registro de corrimiento que al contador. La desviación del reloj puede hacer que el registro de corrimiento se desplace *antes* que el valor a_0 se utilice para que el contador se incremente. Por tanto, tal vez la señal *EB* de la figura 10.11 no logre que el contador se incremente en el flanco apropiado del reloj incluso si el valor de a_0 es 1.

Para la operación adecuada de los circuitos secuenciales síncronos es esencial reducir al mínimo la desviación del reloj. Los chips que contienen muchos flip-flops, como los PLD, usan redes de cables diseñadas meticulosamente para distribuir la señal de reloj a los flip-flops. En la figura 10.44 se ofrece un ejemplo de una red de reloj de distribución. Cada nodo etiquetado *ff* representa la entrada del reloj de un flip-flop; para dar claridad, no se muestran los flip-flops. El buffer en el lado izquierdo de la figura produce la señal de reloj, la cual se distribuye a los flip-flops de tal manera que la longitud del cable entre cada uno de ellos y la fuente de la señal de reloj sea la misma. Debido a la apariencia de las secciones de los cables, que se asemeja a la letra H, la red de distribución del reloj se conoce como *árbol H.* En los PLD el término *reloj global* se refiere a la red del reloj. Un chip PLD por lo general proporciona uno o más relojes globales que pueden conectarse a todos los flip-flops. Cuando se diseña un circuito para implementarlo en un chip como éste, una buena práctica de diseño consiste en conectar todos los flip-flops del circuito a una sola señal de reloj global. La conexión de las compuertas lógicas a las entradas de reloj de los flip-flops, como vimos para el circuito enable de la figura 10.43, debe evitarse.

Es útil poder asegurar que un circuito secuencial se reinicialice en un estado conocido cuando se energiza por primera vez el circuito. Una buena práctica de diseño es conectar las entradas reset (de borrado) asíncronas de todos los flip-flops a una red de cableado que proporcione una señal reset de poca desviación. Los PLD casi siempre proporcionan una red de cableado *reset global* para este propósito.

10.3.2 PARÁMETROS DE SINCRONIZACIÓN DE LOS FLIP-FLOPS

Estudiamos los parámetros de sincronización para elementos de almacenamiento en la sección 7.3.1. Los datos que se van a sincronizar en un flip-flop han de estar estables t_{su} antes del flanco activo del reloj y deben permanecer estables t_h después de éste. Un cambio en el valor de la salida Q

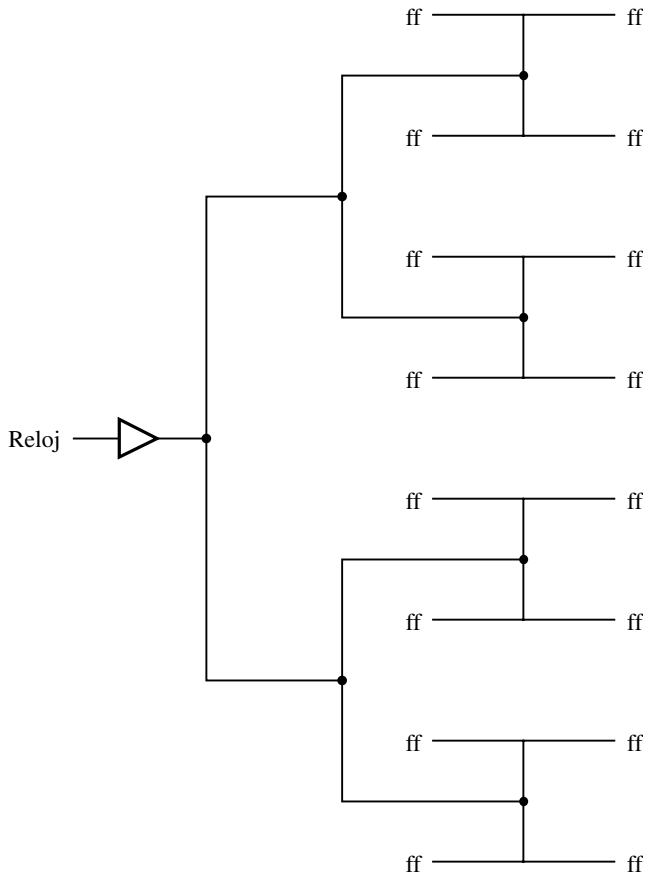


Figura 10.44 Una red de distribución del reloj en árbol H.

aparece después del *retraso de registro* t_{rd}. Se necesita un *tiempo de retraso de salida,* t_{od}, para que el cambio en Q se propague a un pin de salida en el chip. Estos parámetros de sincronización representan el comportamiento de un flip-flop individual sin considerar cómo se conecta éste a otros sistemas de circuitos en un chip de un circuito integrado.

En la figura 10.45 se muestra un flip-flop como parte de un circuito integrado. Las conexiones se muestran desde el reloj del flip-flop, *D,* y las terminales Q a los pines del encapsulado. Hay un buffer de entrada asociado con cada pin del chip. Otro sistema de circuitos también puede estar conectado al flip-flop; el cuadro sombreado representa un circuito combinacional conectado a *D.* Los retrasos de propagación entre los pines del encapsulado y el flip-flop están etiquetados en la figura como t_{Data}, t_{Clock} y t_{od}.

En los sistemas digitales las señales de salida de un chip se utilizan como señales de entrada a otro chip. Con frecuencia los flip-flops en todos los chips se manejan por medio de un reloj común que tiene poca desviación. Las señales deben propagarse desde las salidas Q de los flip-flops en un chip a las entradas *D* de los flip-flops en otro chip. Para asegurar que se cumplen todas las especificaciones de tiempo es preciso considerar los retrasos de salida en un chip y los de entrada en otro.

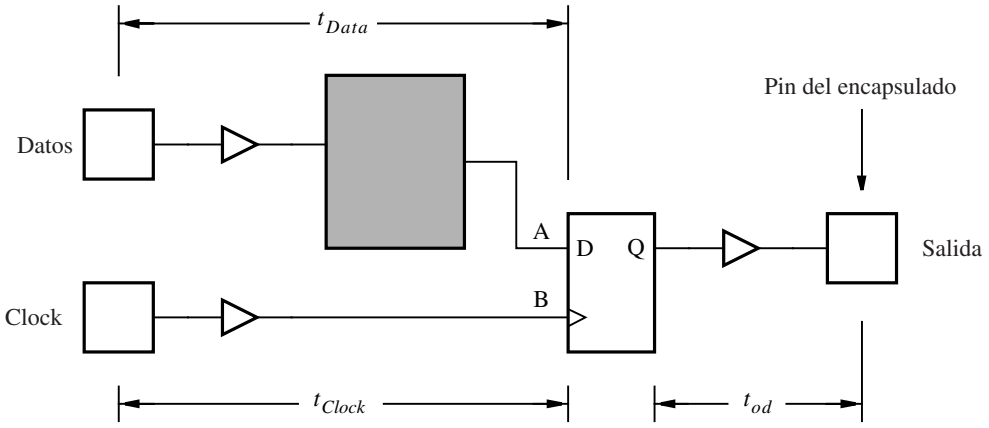


Figura 10.45 Un flip-flop en un chip de circuito integrado.

El retraso t_{co} determina cuánto tiempo pasa desde que ocurre un flanco activo del reloj en el pin del reloj del encapsulado hasta que un cambio en la salida de un flip-flop aparece en un pin de salida en el chip. Ese retraso consta de tres partes principales. La señal de reloj primero debe propagarse desde su pin de entrada en el chip a la entrada del reloj, *Clock,* del flip-flop. Este retraso está etiquetado como t_{Clock} en la figura 10.45. Después del retraso de registro t_{rd}, el flip-flop produce una salida nueva, la cual demora t_{od} en propagarse al pin de salida. Un ejemplo de los parámetros de sincronización tomado de un chip CPLD comercial es $t_{Clock} = 1.5$ ns, $t_{rd} = 1$ ns y $t_{od} = 2$ ns. Estos parámetros proporcionan el retraso a partir del flanco activo del reloj hasta el cambio en el pin de salida como $t_{co} = 4.5$ ns.

Si los chips están separados por una distancia grande, los retrasos de propagación entre ellos han de tenerse en cuenta. Pero en la mayor parte de los casos la distancia entre los chips es pequeña y el tiempo de propagación de las señales entre ellos es insignificante. Una vez que una señal alcanza el pin de entrada en un chip, los valores relativos de t_{Data} y t_{Clock} (véase la figura 10.45) deben considerarse. Por ejemplo, en la figura 10.46 suponemos que $t_{Data} = 4.5$ ns y $t_{Clock} = 1.5$ ns. El tiempo de preparación para los flip-flops en el chip se especifica como $t_{su} = 3$ ns. En la figura la señal *Data* cambia de nivel bajo a alto 3 ns antes del flanco positivo del reloj, lo cual debe satisfacer los requisitos de preparación. La señal *Data* tarda 4.5 ns en llegar al flip-flop,

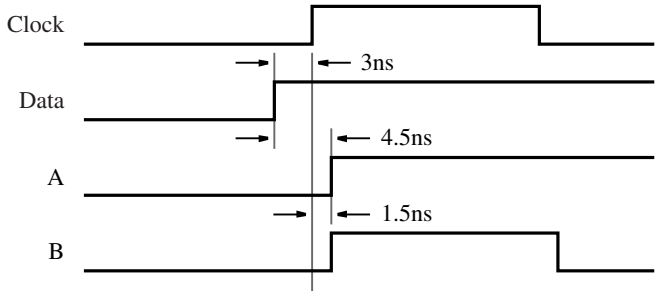


Figura 10.46 Sincronización del flip-flop en un chip.

mientras que la señal *Clock* tarda sólo 1.5 ns. La señal llamada *A* y la señal de reloj llamada *B* llegan al flip-flop al mismo tiempo. El requisito del tiempo de preparación se viola y el flip-flop puede volverse inestable. Para evitar esta condición es necesario aumentar el tiempo de preparación como si se viera desde fuera del chip.

El tiempo de espera para los flip-flops también se ve afectado por los retrasos en el nivel del chip. El resultado es en general una reducción en el tiempo de espera, en vez de un incremento. Por ejemplo, con los parámetros de sincronización de la figura 10.46 asumimos que el tiempo de espera es $t_h = 2$ ns. Supóngase que la señal en el pin *Data* del chip cambia de valor exactamente al mismo tiempo en que ocurre un flanco activo del reloj en el pin *Clock*. El cambio en la señal *Clock* alcanzará el nodo *B* 4.5 − 1.5 = 3 ns antes de que el cambio en *Data* llegue al nodo *A*. Por consiguiente, aun cuando el cambio externo en *Data* coincide con el flanco activo del reloj, el tiempo de espera requerido de 2 ns no se viola.

Para los circuitos grandes, asegurar que los parámetros de sincronización del flip-flop cumplen esto es todo un reto. Tanto esos parámetros como los flip-flops mismos y los retrasos relativos en que incurre el reloj y las señales de datos han de tenerse en cuenta. Los sistemas CAD incluyen herramientas que pueden revisar los tiempos de preparación y espera de todos los flip-flops en forma automática. Esta tarea se realiza utilizando la simulación de tiempo, así como herramientas de propósito especial para análisis de sincronización.

10.3.3 ENTRADAS ASÍNCRONAS A LOS FLIP-FLOPS

En nuestros ejemplos de circuitos secuenciales síncronos hemos supuesto que los cambios en todas las señales de entrada ocurren poco tiempo después de un flanco activo del reloj. Las razones de esta suposición consisten en que las entradas a un circuito se producen como las salidas de otro y la misma señal de reloj se usa para ambos circuitos. En la práctica, algunas de las entradas a un circuito pueden generarse en forma asíncrona respecto a la señal de reloj. Si estas señales se conectan a la entrada *D* de un flip-flop, entonces los tiempos de preparación o de espera pueden violarse.

Cuando los tiempos de preparación o de espera de un flip-flop se violan, la salida del flip-flop puede establecerse en un nivel de voltaje que no corresponde al valor lógico 0 ni de 1. Decimos que el flip-flop se halla en un estado *metaestable*. Con el tiempo el flip-flop se instala en uno de los estados estables, 0 o 1, pero el tiempo requerido para recuperarse del estado metaestable no es predecible. Un enfoque común para trabajar con las entradas asíncronas se ilustra en la figura 10.47. La entrada de datos asíncronos está conectada a un registro de corrimiento de dos bits. La salida del primer flip-flop, etiquetada *A* en la figura, a veces se volverá metaestable. Pero si el periodo del reloj es suficientemente grande, entonces *A* volverá a un valor lógico estable antes que ocurra el siguiente pulso del reloj. Por consiguiente, la salida del segundo flip-flop no se

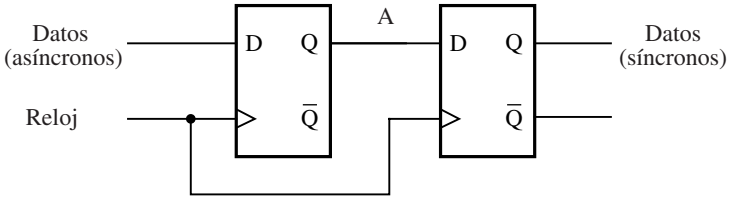


Figura 10.47 Entradas asíncronas.

volverá metaestable y puede conectarse en forma segura a otras partes del circuito. El circuito de sincronización introduce un retraso de un ciclo de reloj antes de que la señal pueda ser utilizada por el resto del circuito.

Los chips comerciales, como los PLD, especifican el periodo de reloj mínimo permisible que debe usarse para el circuito de la figura 10.47 a fin de resolver el problema de metaestabilidad. En la práctica, no es posible *garantizar* que el nodo *A* siempre será estable antes que un flanco del reloj ocurra. La hoja de datos indica una probabilidad de que el nodo *A* sea estable como una función del periodo del reloj. No continuaremos con este tema; el lector interesado puede remitirse a las referencias [10, 11] para un análisis más detallado.

10.3.4 ELIMINACIÓN DE REBOTES EN INTERRUPTORES

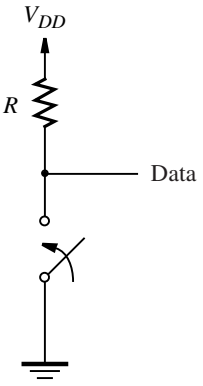
Las entradas a un circuito lógico a veces se generan por medio de interruptores mecánicos. Un problema con estos interruptores es que rebotan cerca de sus puntos de contacto cuando cambian de una posición a otra. En la figura 10.48*a* se muestra un interruptor de un polo y un tiro que proporciona una entrada a un circuito lógico. Si el interruptor está abierto, entonces la señal *Data* tiene el valor 1. Cuando el interruptor se pone en la posición cerrada, *Data* se vuelve 0, pero el interruptor rebota por cierto tiempo, lo que ocasiona que *Data* oscile entre 1 y 0. En general, el rebote persiste alrededor de 10 ms.

No hay una forma sencilla de abordar este problema del rebote usando el interruptor de un polo y un tiro. Si ha de usarse este tipo de interruptor, entonces una solución posible consiste en emplear un circuito, por ejemplo un contador, para medir un retraso largo en forma adecuada hasta que el rebote se detenga (véase el problema 10.23).

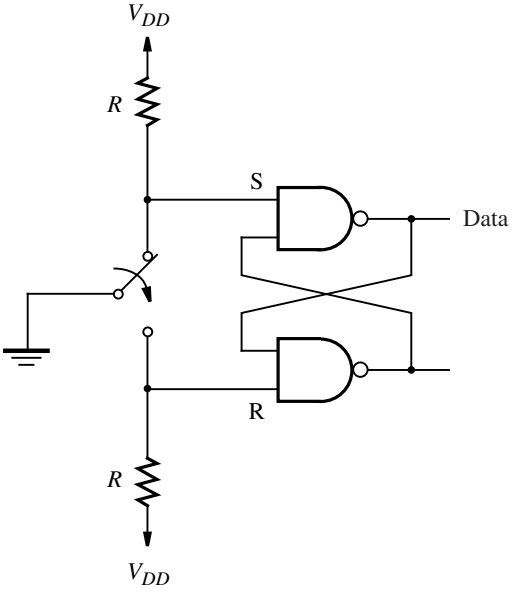
Un mejor enfoque para resolver el rebote del interruptor se muestra en la figura 10.48*b*. Este método utiliza un interruptor de un polo y doble tiro y un latch SR básico para generar una entrada a un circuito lógico. Cuando el interruptor está en la posición inferior, la entrada *R* en el latch es 0 y $Data = 0$. Cuando el interruptor se pone en la posición superior, la entrada *S* en el latch se vuelve 0, lo cual establece *Data* en 1. Si el interruptor rebota desde la posición superior, las entradas al latch se vuelven $R = S = 1$ y el latch almacena el valor $Data = 1$. Cuando el interruptor se pone en la posición inferior, *Data* cambia a 0 y este valor se almacena en el latch si el interruptor rebota. Nótese que cuando un interruptor rebota, no puede hacerlo completamente entre las terminales *S* y *R*; sólo rebota alejándose un poco de una de las terminales y luego regresa a ella.

10.4 COMENTARIOS FINALES

En este capítulo se brindaron varios ejemplos de sistemas digitales que incluyen una o más FSM, así como bloques de construcción como sumadores, registros, registros de corrimiento y contadores. Hemos mostrado cómo las cartas ASM pueden utilizarse para facilitar el diseño de un sistema digital y hemos mostrado cómo los circuitos pueden describirse mediante código de VHDL. También estudiamos una serie de problemas prácticos, como la desviación del reloj, la sincronización de las entradas asíncronas y la eliminación del rebote en interruptores. Algunos libros notables que también cubren el material presentado en este capítulo incluyen [3-10].



a) Interruptor de un polo y un tiro



b) Interruptor de un polo y doble tiro con un latch SR básico

Figura 10.48 Circuito de eliminación de rebotes en interruptores.

PROBLEMAS

10.1 El circuito de la figura 10.4 proporciona un registro de corrimiento en el que la entrada de control de carga en paralelo es independiente de la entrada enable. Muestre un circuito de registro de corrimiento distinto en el que la operación de carga en paralelo pueda realizarse sólo cuando la entrada enable también se valide.

10.2 La carta ASM de la figura 10.10, que describe el circuito de conteo de bits, incluye salidas tipo Moore en los estados $S1$, $S2$ y $S3$, y tiene una salida tipo Mealy en el estado $S2$.
a) Muestre cómo puede modificarse la carta ASM de modo que tenga sólo salidas tipo Moore en el estado $S2$.
b) Dé la carta ASM para el circuito de control correspondiente al inciso *a*).
c) Proporcione el código de VHDL que represente el circuito de control modificado.

10.3 En la figura 10.17 se muestra el circuito de trayectoria de datos para el multiplicador de corrimiento y suma. Use un registro de corrimiento para B de modo que b_0 sirva para decidir si A debe o no sumarse a P. Un enfoque distinto consiste en utilizar un registro normal para almacenar el operando B y usar un contador y un multiplexor para seleccionar el bit b_i en cada etapa de la operación de multiplicación.
a) Muestre la carta ASM que utiliza un registro normal para B, en vez de un registro de corrimiento.
b) Muestre el circuito de trayectoria de datos que corresponde al inciso *a*).
c) Proporcione la carta ASM para el circuito de control correspondiente al inciso *b*).
d) Dé el código de VHDL que representa el circuito multiplicador.

10.4 Escriba código de VHDL para el circuito divisor que tiene el circuio de trayectoria de datos de la figura 10.23 y el circuito de control representado por la carta ASM de la figura 10.24.

10.5 En la sección 10.2.4 mostramos cómo implementar la división larga tradicional hecha "a mano". Un enfoque distinto para implementar la división de enteros es realizar una resta repetida como se indica en el seudocódigo de la figura P10.1.

```
Q = 0 ;
R = A ;
while ((R − B) > 0) do
      R = R − B ;
      Q = Q + 1 ;
end while ;
```

Figura P10.1 Seudocódigo para la división de enteros.

a) Proporcione una carta ASM que represente el seudocódigo de la figura P10.1.
b) Muestre el circuito de trayectoria de datos que corresponde al inciso *a*).
c) Dé la carta ASM para el circuito de control correspondiente al inciso *b*).
d) Proporcione el código de VHDL que representa el circuito divisor.
e) Comente los méritos relativos y los inconvenientes de su circuito en comparación con el circuito diseñado en la sección 10.2.4.

10.6 En la carta ASM de la figura 10.32, los dos estados *S*3 y *S*4 se usan para calcular la media $M = Sum/k$. Muestre una carta ASM modificada que combine los estados *S*3 y *S*4 en un solo estado, llamado *S*3.

10.7 Escriba código de VHDL para la FSM representada por su carta ASM definida en el problema 10.6.

10.8 En la carta ASM de la figura 10.36, especificamos la asignación $C_j \leftarrow C_i$ en el estado *S*2, y luego en el estado *S*3 incrementamos C_j en 1. ¿Es posible eliminar el estado *S*3 si la asignación $C_j \leftarrow C_i + 1$ se realiza en *S*2? Explique cualesquiera implicaciones que este cambio tenga sobre los circuitos de control y de trayectoria de datos.

10.9 En la figura 10.35 se proporciona el seudocódigo para la operación de ordenación en la que los registros que se están ordenando se indizan utilizando las variables *i* y *j*. En la carta ASM de la figura 10.36, las variables *i* y *j* se implementan utilizando los contadores C_i y C_j. Un enfoque diferente consiste en implementar *i* y *j* mediante dos registros de corrimiento.
a) Rediseñe el circuito para la operación de ordenación utilizando los registros de corrimiento en vez de los contadores para indizar los registros $R_0,\ldots, R_3$.
b) Proporcione el código de VHDL para el circuito diseñado en el inciso *a*).
c) Comente los méritos relativos y los inconvenientes de su circuito en comparación con el circuito que utiliza los contadores C_i y C_j.

10.10 En la figura 10.42 se muestra un circuito de trayectoria de datos para la operación de ordenación que utiliza buffers triestado para tener acceso a los registros. Usando una herramienta de captura esquemática trace el esquema de la figura 10.42. Construya los demás subcircuitos necesarios utilizando código de VHDL y cree símbolos gráficos que los representen. Describa el circuito de control mediante código de VHDL, diseñe un símbolo gráfico para él y conecte este símbolo a los módulos de trayectoria de datos del esquema. Proporcione el resultado de una simulación para su circuito implementado en un chip de su elección. Lea los apéndices *B*, *C* y *D* para obtener instrucciones respecto al uso de las herramientas CAD.

10.11 En la figura 10.40 se presenta el código de VHDL para el circuito de ordenación. Muestre cómo modificarlo para utilizar un subcircuito que represente un bloque de SRAM de $k \times n$. Utilice el módulo *lpm_ram_dq* para el bloque de SRAM. Elija la opción SRAM síncrona de modo que todos los cambios en el contenido de la SRAM estén sincronizados con la señal de reloj. (*Sugerencia*: Utilice el complemento de la señal de reloj para sincronizar las operaciones SRAM, ya que este método permite usar sin cambios el código de VHDL para la FSM mostrado en la figura 10.40.)

10.12 Diseñe un circuito que encuentre el log_2 de un operando almacenado en un registro de *n* bits. Muestre todos los pasos del proceso de diseño y plantee las suposiciones hechas. Proporcione el código de VHDL que describa su circuito.

10.13 En la figura 10.33 se muestra un esquema para el circuito que calcula la operación de la media. Escriba código de VHDL que represente este circuito. Use un arreglo de registros en vez de un bloque de SRAM. Para el subcircuito *divisor*, utilice una operación de corrimiento que divida entre cuatro, en vez de usar el circuito divisor diseñado en la sección 10.2.4.

10.14 El circuito diseñado en la sección 10.2.5 utiliza un sumador para calcular la suma del contenido de los registros. El subcircuito divisor empleado para calcular $M = Sum/k$ también incluye un

sumador. Muestre cómo puede rediseñarse el circuito de modo que contenga un solo subcircuito sumador simple que se use tanto para la operación de suma como para la de división. Muestre sólo el sistema de circuitos adicional requerido para conectarse al sumador; explique su operación.

10.15 Proporcione el código de VHDL para el circuito diseñado en el problema 10.14, incluidos tanto el circuito de trayectoria de datos como el de control.

10.16 El seudocódigo para la operación de ordenación dado en la figura 10.35 utiliza los registros *A* y *B* para almacenar el contenido de los registros que se están ordenando. Muestre el seudocódigo para la operación de ordenación que usa sólo el registro *A* para almacenar los datos temporalmente durante esa operación. Proporcione la carta ASM correspondiente que represente los circuitos de trayectoria de datos y de control requeridos. Emplee multiplexores para interconectar los registros, en el estilo mostrado en la figura 10.37. Dé una carta ASM separada que represente el circuito de control.

10.17 Proporcione el código de VHDL para el circuito de ordenación diseñado en el problema 10.16.

10.18 En la sección 7.14.1 mostramos un sistema digital con tres registros, *R*1 a *R*3, y diseñamos un circuito de control que puede usarse para intercambiar el contenido de los registros *R*1 y *R2.* Proporcione una carta ASM que represente este sistema digital y la operación de intercambio.

10.19 *a*) Para la carta ASM derivada en el problema 10.18, muestre otra carta ASM que especifique las señales de control requeridas para controlar el circuito de trayectoria de datos. Suponga que se usan multiplexores para implementar el bus que conecta los registros, como se muestra en la figura 7.60.
b) Escriba el código de VHDL completo para el sistema del problema 10.18, incluido el circuito de control descrito en el inciso *a*).
c) Sintetice un circuito a partir del código de VHDL escrito en el inciso *b*) y muestre una simulación de tiempo que ilustre la funcionalidad correcta del circuito.

10.20 En la sección 7.14.2 mostramos el diseño de un circuito que funciona como un procesador. Proporcione una carta ASM que describa la funcionalidad de este procesador.

10.21 *a*) Para la carta ASM derivada en el problema 10.20, muestre otra carta ASM que especifique las señales de control requeridas para controlar el circuito de trayectoria de datos en el procesador. Suponga que los multiplexores se usan para implementar el bus que conecta los registros *R*0 a *R*3, en el procesador.
b) Escriba el código de VHDL completo para el sistema del problema 10.20, incluido el circuito de control descrito en el inciso *a*).
c) Sintetice un circuito a partir del código de VHDL escrito en el inciso *b*) y muestre una simulación de tiempo que ilustre la funcionalidad correcta del circuito.

10.22 Considere el diseño de un circuito que controla los semáforos en la intersección de dos calles. El circuito genera las salidas *G*1, *Y*1, *R*1 y *G*2, *Y*2*, R*2, que representan los estados de las luces verde, amarilla y roja, respectivamente, en cada calle. Una luz se enciende si la señal de salida correspondiente tiene el valor 1. Las luces deben controlarse de la manera siguiente: cuando *G*1 se enciende debe permanecer así durante un periodo llamado t_1 y luego apagarse. El apagado de *G*1 debe dar como resultado que *Y*1 se encienda de inmediato; esta luz debe permanecer encendida durante un periodo llamado t_2 y luego apagarse. Cuando *G*1 o *Y*1 estén encendidas, *R*2 debe estar encendida y *G*2 y *Y*2 deben estar apagadas. El apagado de *Y*1 debe hacer que *G*2 se encienda inmediatamente durante el periodo t_1. Cuando *G*2 se apaga, *Y*2 se enciende durante el periodo t_2. Desde luego, cuando *G*2 o *Y*2 estén encendidas, *R*1 debe estar encendida y *G1* y *Y*1 deben estar apagadas.

a) Proporcione una carta ASM que describa el controlador del semáforo. Suponga que existen dos contadores descendentes, uno que sirve para medir el retraso t_1 y otro que se usa para medir t_2. Cada contador tiene entradas de carga en paralelo y enable. Estas entradas se emplean para cargar un valor apropiado que represente ya sea el retraso t_1 o el t_2 y luego permiten que el contador cuente en forma descendente hasta 0.
b) Proporcione una carta ASM para el circuito de control para el controlador del semáforo.
c) Escriba el código de VHDL completo para el controlador del semáforo, incluido el circuito de control del inciso *a*) y los contadores para representar t_1 y t_2. Use cualquier frecuencia de reloj conveniente para sincronizar el circuito y suponga valores de conteo prácticos que representen a t_1 y t_2. Proporcione los resultados de la simulación que ilustran la operación de su circuito.

10.23 Suponga que necesita emplear un interruptor de un polo y un tiro como el mostrado en la figura 10.48*a.* Muestre cómo puede utilizarse un contador como un medio de eliminar el rebote de la señal *Data* producido por el interruptor. (*Sugerencia*: Diseñe una FSM que tenga *Data* como una entrada y produzca la salida *z,* la cual es la versión sin rebote de *Data.* Suponga que tiene acceso a una señal de entrada de reloj con la frecuencia 102.4 kHz, que puede usarse según se requiera.)

10.24 Las señales de reloj comúnmente se generan utilizando chips de propósito especial. Un ejemplo de este tipo de chips es el temporizador programable 555, que se representa en la figura P10.2. Al elegir valores particulares para las resistencias R_a y R_b, así como para el capacitor C_1, el sincronizador 555 puede utilizarse para producir la señal de reloj buscada. Es posible elegir tanto el periodo de la señal de reloj como su ciclo de trabajo. El término *ciclo de trabajo* se refiere al porcentaje del periodo del reloj para el que la señal se halla en nivel alto. Las ecuaciones siguientes

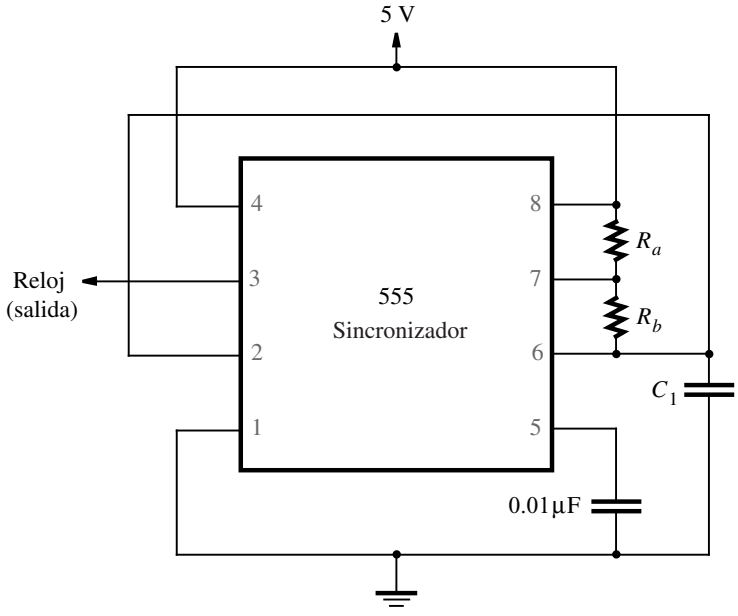


Figura P10.2 El chip del sincronizador programable *555*

definen la señal de reloj producida por el chip

$$\text{Periodo del reloj} = 0.7(R_a + 2R_b)C_1$$

$$\text{Ciclo de trabajo} = \frac{R_a + R_b}{R_a + 2R_b}$$

a) Determine los valores de R_a, R_b y C_1 necesarios para producir una señal de reloj con un ciclo de trabajo de 50% y una frecuencia aproximada de 500 kHz.
b) Repita el inciso *a*) para un ciclo de trabajo de 75%.

BIBLIOGRAFÍA

1. V. C. Hamacher, Z. G. Vranesic y S. G. Zaky, *Computer Organization,* 5a. ed. (McGraw-Hill: Nueva York, 2002).
2. D. A. Patterson y J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface,* 2a. ed. (Morgan Kaufmann: San Francisco, CA, 1998).
3. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano y C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One* (Wiley: Nueva York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, 1994).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4a. ed. (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1999).
11. C. J. Myers, *Asynchronous Circuit Design* (Wiley: Nueva York, 2001).

capítulo

11

PRUEBAS DE LOS CIRCUITOS LÓGICOS

OBJETIVOS DEL CAPÍTULO

En este capítulo se estudian los temas siguientes:

- Varias técnicas para probar los circuitos digitales
- La representación de fallas típicas en un circuito
- La derivación de las pruebas requeridas para probar el comportamiento de un circuito
- El diseño de circuitos para una aplicación fácil de las pruebas

En capítulos anteriores estudiamos el diseño de los circuitos lógicos. Tras seguir un procedimiento de diseño sólido, esperamos que el circuito diseñado tenga el desempeño requerido. Pero ¿cómo se comprueba que el circuito final cumple en realidad los objetivos de diseño? Resulta esencial determinar que el circuito tiene el comportamiento funcional buscado y que satisface cualquier restricción de tiempo impuesta al diseño. Hemos expuesto los aspectos de la sincronización en varias partes del libro. En este capítulo estudiaremos algunas técnicas de aplicación de pruebas que pueden utilizarse para verificar la funcionalidad de un circuito.

Hay varias razones para probar un circuito lógico. Cuando éste se desarrolla por vez primera, es necesario revisar que lo que se diseña cumple las especificaciones funcionales y de tiempo deseadas. Cuando se fabrican varias copias de un circuito diseñado correctamente es indispensable probar cada una para tener la seguridad de que no se han introducido fallas en el proceso de manufactura. También es preciso probar los circuitos empleados en el equipo instalado en el área cuando se sospecha que puede haber algo erróneo.

La base de todas las técnicas de prueba consiste en aplicar conjuntos predefinidos de entradas, llamados *pruebas,* a un circuito y comparar las salidas observadas con los patrones que se supone produce un circuito que funciona bien. El reto es derivar un número relativamente pequeño de pruebas que brinden una indicación adecuada de que el circuito es correcto. El enfoque exhaustivo de aplicar todas las pruebas posibles es poco práctico para los circuitos grandes, ya que existen demasiadas pruebas posibles.

11.1 MODELO DE FALLAS

Un circuito funciona mal cuando hay algo erróneo en él, como una falla en el transistor o en el cableado de interconexión. Muchas cosas pueden salir mal y desembocar en diversas fallas. Un transistor interruptor puede fallar de modo que quede permanentemente cerrado o abierto. Un cable del circuito puede acortarse a V_{DD} o a tierra, o simplemente romperse. Tal vez exista una conexión indeseada entre dos cables. Una compuerta lógica puede generar una señal de salida errónea debido a una falla en el sistema de circuitos que la implementa. Lidiar con diferentes tipos de fallas es molesto. Por suerte, es posible restringir el proceso de pruebas a fallas simples y obtener resultados generalmente satisfactorios.

11.1.1 MODELO DE ATASCAMIENTO *(STUCK-AT)*

La mayor parte de los circuitos analizados en este texto utiliza las compuertas lógicas como bloques de construcción básicos. Un buen modelo para representar fallas en este tipo de circuitos consiste en suponer que todas se manifiestan como algunos cables (entradas o salidas de compuertas) atascados permanentemente en el valor lógico 0 o 1. Indicamos que un cable, *w,* tiene una señal indeseable que siempre corresponde al valor lógico 0 al decir que *w* está *atascado en 0* (*stuck-at-0*), lo cual se indica como *w*/0. Si *w* tiene una señal indeseable que siempre es igual al valor lógico 1, entonces está *atascado en 1,* lo que se indica como *w*/1.

Un ejemplo claro de una falla de atascamiento es cuando una entrada a una compuerta se conecta incorrectamente al suministro de corriente, ya sea a V_{DD} o a tierra. Pero el modelo de atascamiento también es útil para lidiar con fallas de otros tipos, las cuales suelen provocar los mismos problemas que los que se tienen cuando un cable se queda atascado en un valor lógico en particular. El impacto exacto de una falla en el sistema de circuitos que implementa la compuerta lógica depende de la tecnología usada. Restringiremos la atención a las fallas de atascamiento y examinaremos el proceso de pruebas suponiendo que son las únicas fallas que pueden ocurrir.

11.1.2 Fallas individuales y múltiples

Un circuito puede estar defectuoso a causa de una o a varias fallas. Tratar con varias es difícil porque cada una puede ocurrir de muchas maneras. Un enfoque pragmático consiste en considerar sólo las fallas individuales. La práctica ha demostrado que un conjunto de pruebas que sirven para detectar todas las fallas individuales en un circuito también sirven para determinar la vasta mayoría de las fallas múltiples.

Una falla se detecta si el valor de salida producido por el circuito defectuoso es diferente del producido por el circuito en buen estado cuando se aplica una prueba apropiada como entrada. Se supone que cada prueba puede detectar la ocurrencia de una o más fallas. Un conjunto completo de pruebas utilizado para un circuito se conoce como *conjunto de pruebas.*

11.1.3 Circuitos CMOS

Los circuitos lógicos CMOS presentan ciertos problemas especiales en términos de comportamiento incorrecto. Los transistores pueden fallar quedando en un estado permanentemente abierto o reducido (cerrado). Muchas de estas fallas se manifiestan como fallas de atascamiento. Pero algunas producen un comportamiento totalmente distinto. Por ejemplo, los transistores que presentan errores en el estado reducido pueden causar un flujo continuo de corriente de V_{DD} a tierra, lo cual puede crear un voltaje de salida intermedio que tal vez no esté determinado como un 0 o un 1 lógico. Los transistores que presentan fallas en el estado abierto pueden llevar a condiciones en las que el capacitor de salida conserva su nivel de carga porque el interruptor que se supone lo va a descargar está roto. El resultado es que un circuito CMOS combinacional comienza a comportarse como un circuito secuencial.

Las técnicas específicas para probar los circuitos CMOS están más allá del ámbito de este libro. Un estudio preliminar del tema puede encontrarse en las referencias [1-3]. La aplicación de pruebas a los circuitos CMOS ha sido objeto de una investigación considerable [4-6]. Supondremos que un conjunto de pruebas desarrollado usando el modelo de atascamiento proporcionará una cobertura adecuada de las fallas en todos los circuitos.

11.2 Complejidad de un conjunto de pruebas

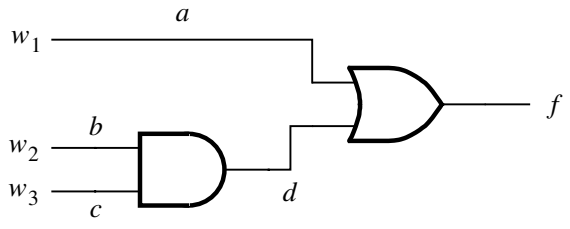
Hay una gran diferencia en probar los circuitos combinacionales y los secuenciales. Los primeros pueden probarse en forma adecuada independientemente de su diseño. Los circuitos secuenciales ofrecen un reto mayor, ya que el comportamiento de un circuito bajo prueba está influido no sólo por las pruebas que se aplican a las entradas externas sino también por los estados en que se halla el circuito cuando se hacen las pruebas. Es muy difícil probar un circuito secuencial creado por un diseñador que no tomó en cuenta la facilidad para aplicarle pruebas. Sin embargo, es posible diseñar estos circuitos para hacer que la aplicación de pruebas sea más sencilla, como veremos en la sección 11.6. Comenzaremos por considerar la aplicación de pruebas a los circuitos combinacionales.

Una forma obvia de probar un circuito combinacional consiste en aplicar un conjunto de pruebas que comprenda todas las combinaciones de entrada posibles. Por tanto, sólo es necesario revisar si los valores de salida producidos por el circuito son los mismos que los especificados en una tabla de verdad que defina al circuito. Este enfoque funciona bien para circuitos pequeños,

donde el conjunto de pruebas no es grande, pero se vuelve totalmente impráctico para circuitos grandes con muchas variables de entrada. Por suerte, no es necesario aplicar todas las 2^n combinaciones a un circuito de n entradas como pruebas. Un conjunto de pruebas completo, capaz de detectar todas las fallas individuales, en general comprende un número de pruebas mucho menor.

En la figura 11.1*a* se muestra un circuito simple de tres entradas para el cual queremos determinar el conjunto de pruebas más pequeño. Un conjunto de pruebas exhaustivo incluiría las ocho combinaciones de entrada. Este circuito consta de cinco cables, etiquetados en la figura como *a, b, c, d* y *f*. Al usar nuestro modelo de fallas, cada cable puede atascarse en 0 o en 1.

En la figura 11.1*b* se enumera la utilidad de las ocho combinaciones como pruebas posibles para el circuito. La combinación $w_1w_2w_3 = 000$ puede detectar la ocurrencia de una falla de atascamiento en 1 en los cables *a, d* y *f*. En un buen circuito esta prueba da como resultado la salida $f = 0$. Sin embargo, si ocurre cualquiera de las fallas *a*/1, *d*/1 o *f*/1, el circuito producirá $f = 1$ cuando se aplique la combinación de entrada 000. La prueba 001 provoca que $f = 0$ en un



a) Circuito

Prueba $w_1w_2w_3$	Falla detectada									
	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$f/0$	$f/1$
000		√						√		√
001		√		√				√		√
010		√				√		√		√
011			√		√		√		√	
100	√								√	
101	√								√	
110	√								√	
111									√	

b) Fallas detectadas por las diversas combinaciones de entrada

Figura 11.1 Detección de fallas en un circuito simple.

circuito con buen funcionamiento, y da como resultado $f = 1$ si cualquiera de las fallas $a/1$, $b/1$, $d/1$ o $f/1$ ocurre. Esta prueba puede detectar la ocurrencia de cuatro fallas distintas. Se dice que *cubre* estas fallas. La última prueba, 111, puede detectar sólo una falla, $f/0$.

Un conjunto de pruebas mínimo que cubre todas las fallas en el circuito puede obtenerse a partir de la tabla por inspección. Algunas fallas sólo están cubiertas por una prueba, lo que significa que las pruebas de este tipo deben incluirse en el conjunto de pruebas. La falla $b/1$ se cubre sólo con 001. La falla $c/1$ se cubre únicamente con 010. Las fallas $b/0$, $c/0$ y $d/0$ se cubren sólo con 011. Por tanto, estas tres pruebas son esenciales. Para las fallas restantes pueden utilizarse diferentes pruebas. La selección de las pruebas 001, 010 y 011 cubre todas las fallas excepto $a/0$, la cual puede cubrirse por medio de tres pruebas distintas. Al elegir 100 de manera arbitraria, un conjunto de pruebas completo para el circuito es

$$\text{Conjunto de pruebas} = \{001, 010, 011, 100\}$$

La conclusión es que todas las fallas de atascamiento posibles en este circuito pueden detectarse usando cuatro pruebas, en vez de las ocho que se ocuparían si simplemente tratáramos de probar el circuito mediante su tabla de verdad completa.

El tamaño del conjunto de pruebas completo para un circuito de n entradas es en general mucho menor que 2^n. Pero este tamaño aún puede ser inaceptablemente grande en términos prácticos. Además, es probable que obtener el conjunto de pruebas mínimo sea una tarea abrumadora incluso para los circuitos de tamaño moderado. Sin duda, el enfoque sencillo de la figura 11.1 no resulta práctico; en la sección siguiente exploraremos uno más interesante.

11.3 SENSIBILIZACIÓN DE TRAYECTORIAS

Obtener un conjunto de pruebas al considerar las fallas individuales en todos los cables de un circuito, como se hizo en la sección 11.2, no es atractivo desde un punto de vista práctico. Hay demasiados cables y demasiadas fallas por tomar en cuenta. Una mejor opción consiste en considerar varios cables que forman una ruta como una entidad que puede probarse en busca de varias fallas mediante una sola prueba. Es posible activar una trayectoria de modo que los cambios en la señal que se propaga a lo largo de ella tengan un impacto directo en la señal de salida.

En la figura 11.2 se ilustra una trayectoria desde la entrada w_1 a la salida f, a través de tres compuertas, las cuales se componen de los cables a, b, c y f. La trayectoria se activa al asegurar que las otras trayectorias del circuito no determinan el valor de la salida f. Por tanto, la entrada w_2 debe establecerse en 1 de modo que la señal en b dependa sólo del valor en a. La entrada w_3 debe

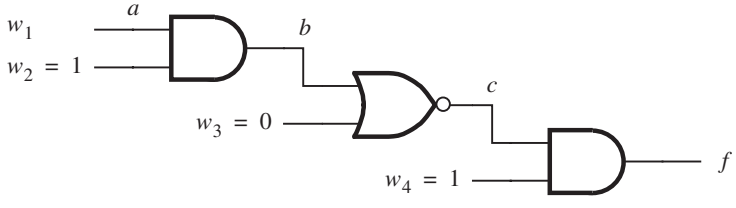


Figura 11.2 Una trayectoria sensibilizada.

ser 0 de manera que no afecte a la compuerta NOR, y w_4 debe ser 1 para no afectar la compuerta AND. Luego, si $w_1 = 0$ la salida será $f = 1$, mientras que $w_1 = 1$ ocasionará que $f = 0$. En vez de decir que la trayectoria de w_1 a f está activada, en la bibliografía técnica se usa un término más específico: se dice que la ruta está *sensibilizada.*

Para sensibilizar una trayectoria a través de una entrada de una compuerta AND o NAND, todas las demás entradas deben establecerse en 1. Para sensibilizar una trayectoria por una entrada de una compuerta OR o NOR, todas las demás entradas deben ser 0.

Considérese ahora el efecto de las fallas a lo largo de una trayectoria sensibilizada. La falla $a/0$ de la figura 11.2 causará que $f = 1$ incluso si $w_1 = 1$. El mismo efecto ocurre si las fallas $b/0$ o $c/1$ están presentes. Por ende, la prueba $w_1w_2w_3w_4 = 1101$ detecta la ocurrencia de las fallas $a/0$, $b/0$ y $c/1$. De forma similar, si $w_1 = 0$, la salida debe ser $f = 1$. Pero si cualquiera de las fallas $a/1$, $b/1$ o $c/0$ se presenta, la salida será $f = 0$. Por consiguiente, estas tres fallas son detectables con la prueba 0101. La presencia de cualquier falla de atascamiento a lo largo de la trayectoria sensibilizada es detectable por la aplicación únicamente de dos pruebas.

Es probable que el número de trayectorias en un circuito sea mucho más pequeño que el de cables individuales. Esto sugiere que podría ser atractivo derivar un conjunto de pruebas basado en las trayectorias sensibilizadas. Esta posibilidad se ilustra en el ejemplo siguiente.

Ejemplo 11.1 **PRUEBAS DE TRAYECTORIA SENSIBILIZADA** Considere el circuito de la figura 11.3, el cual tiene cinco trayectorias. La trayectoria $w_1 - c - f$ se sensibiliza al establecer $w_2 = 1$ y $w_4 = 0$. No importa si w_3 es 0 o 1, ya que $w_2 = 1$ hace que la señal en el cable b sea igual a 0, lo cual fuerza a que $d = 0$ independientemente del valor de w_3. Por tanto, la trayectoria está sensibilizada al establecer $w_2w_3w_4 = 1x0$, donde el símbolo x significa que el valor de w_3 no importa. Ahora las pruebas $w_1w_2w_3w_4 = 01x0$ y $11x0$ detectan todas las fallas a lo largo de esta trayectoria. La segunda trayectoria, $w_2 - c - f$, se prueba usando 1000 y 1100. La trayectoria $w_2 - b - d - f$ se prueba utilizando 0010 y 0110. Las pruebas para la trayectoria $w_3 - d - f$ son x000 y x010. La quinta trayectoria, $w_4 - f$, se prueba con 0x00 y 0x01. En vez de utilizar estas 10 pruebas, podemos observar que la prueba 0110 sirve también como la prueba 01x0, la prueba 1100 sirve además como 11x0, la prueba 1000 como x000 y la prueba 0010 como x010. Por ende, el conjunto de pruebas completo es

$$\text{Conjunto de pruebas} = \{0110, 1100, 1000, 0010, 0x00, 0x01\}$$

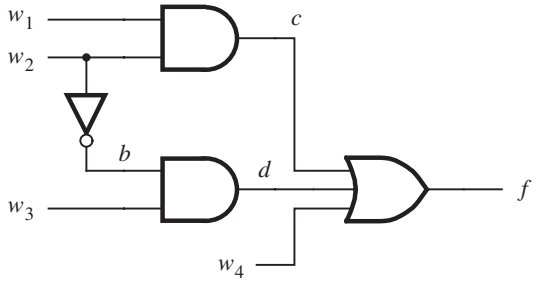


Figura 11.3 Circuito para el ejemplo 11.1.

Si bien este enfoque es más simple, aún resulta impráctico para los circuitos grandes. Sin embargo, el concepto de sensibilización de trayectorias es muy útil, como veremos en el análisis siguiente.

11.3.1 DETECCIÓN DE UNA FALLA ESPECÍFICA

Supóngase que sospechamos que el circuito de la figura 11.3 tiene una falla en la que el cable *b* está atascado en 1. Una prueba que determina la presencia de esta falla puede obtenerse al sensibilizar una trayectoria que propaga el efecto de la falla hacia la salida, *f*, donde puede observarse. La trayectoria va de *b* a *d* y luego a *f*. Es preciso establecer $w_3 = 1$, $w_4 = 0$ y $c = 0$. Esto último puede lograrse al establecer $w_1 = 0$. Si *b* está atascado en 1, entonces es necesario aplicar una entrada que normalmente produciría el valor de 0 en el cable *b,* de modo que los valores de salida de los circuitos en buen estado y defectuosos sean distintos. Por consiguiente, w_2 debe establecerse en 1. Así, la prueba que detecta la falla *b*/1 es $w_1w_2w_3w_4 = 0110$.

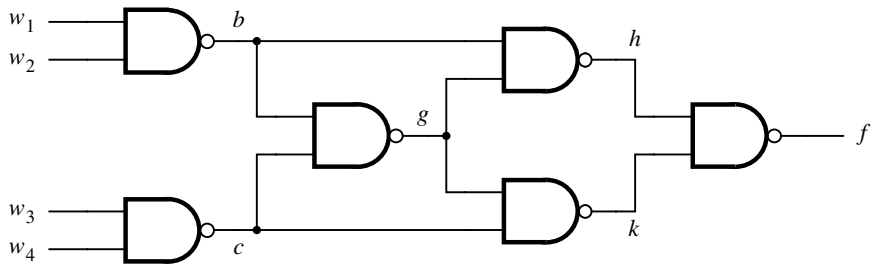
En general, la falla en cierto cable puede detectarse al propagar el efecto de la falla a la salida, sensibilizando una trayectoria apropiada. Esto implica asignar valores a otras entradas de las compuertas a lo largo de la trayectoria. Estos valores deben obtenerse al asignar valores específicos a las entradas primarias, lo cual tal vez no siempre sea posible. En el ejemplo 11.2 se ilustra el proceso.

Ejemplo 11.2

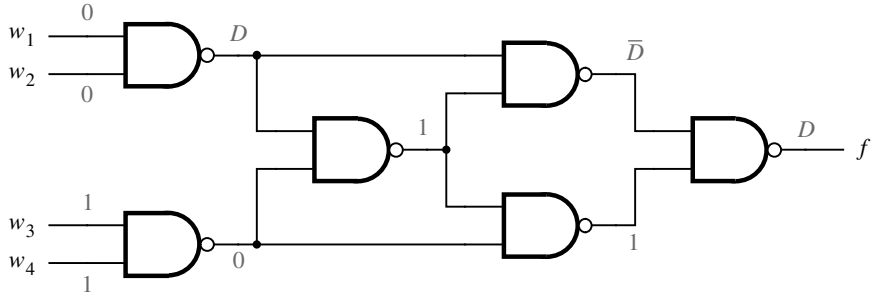
PROPAGACIÓN DE FALLAS A medida que el efecto de una falla se propaga a través de las compuertas a lo largo de la trayectoria sensibilizada, la polaridad de la señal cambiará cuando pasa por una compuerta inversora. Sea el símbolo *D* una falla general de atascamiento en 0. El efecto de esta falla permanecerá sin alteraciones cuando pase por una compuerta AND u OR. Si *D* está en una entrada de una compuerta AND (OR) y las otras entradas se establecen en 1 (0), entonces la salida de la compuerta se comportará como si tuviera *D* en ella. Pero si *D* está en una entrada de una compuerta NOT, NAND o NOR, entonces la salida aparecerá atascada en 1, lo que se indica como $\overline{D}$.

En la figura 11.4 se muestra cómo el efecto de una falla puede propagarse utilizando los símbolos *D* y $\overline{D}$. Suponga primero que en el cable *b* hay una falla de atascamiento en 0*;* es decir, *b*/0. Queremos propagar el efecto de esta falla a lo largo de la trayectoria $b - h - f$, lo cual puede hacerse como se indica en la figura 11.4*b.* El establecer $g = 1$ propaga la falla al cable *h*. Entonces *h* aparece atascado en 1, lo cual se indica por medio de $\overline{D}$. Enseguida el efecto se propaga a *f* al establecer $k = 1$. Como la última compuerta NAND también invierte la señal, la salida se vuelve igual a *D,* lo que equivale a *f*/0. Por tanto, en un circuito en buen estado la salida debe ser 1, pero en un circuito defectuoso será 0. A continuación debemos determinar que es posible tener $g = 1$ y $k = 1$ al asignar los valores apropiados a las variables de entrada principales. A esto se le llama *revisión de consistencia.* Al establecer $c = 0$, tanto *g* como *k* se verán forzadas a tomar el valor de 1, lo cual puede lograrse con $w_3 = w_4 = 1$. Finalmente, para provocar la propagación de la falla *D* en el cable *b* es necesario aplicar una señal que ocasione que *b* tenga el valor 1, lo que significa que w_1 o w_2 deben ser 0. Así, la prueba $w_1w_2w_3w_4 = 0011$ detecta la falla *b*/0.

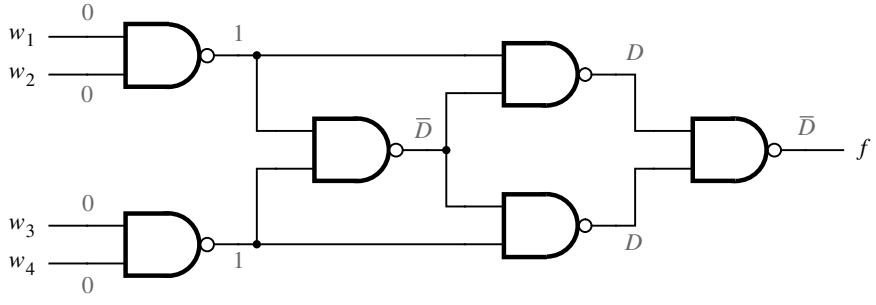
Suponga ahora que el cable *g* está atascado en 1, lo cual se indica por medio de $\overline{D}$. Podemos tratar de propagar el efecto de esta falla a través de la trayectoria $g - h - f$ al establecer $b = 1$ y



a) Circuito



b) Detección de la falla $b/0$



c) Detección de la falla $g/1$

Figura 11.4 Detección de fallas.

$k = 1$. Para hacer que $b = 1$, establecemos $w_1 = w_2 = 0$. Para lograr que $k = 1$, debemos hacer que $c = 0$. Pero también es necesario hacer que la propagación de la falla $\overline{D}$ en g por medio de una señal que haga que $g = 0$ en el circuito en buen estado. Esto puede hacerse sólo si $b = c = 1$. El problema es que al mismo tiempo necesitamos que $c = 0$, para hacer que $k = 1$. Por consiguiente, la revisión de consistencia fracasa y la falla $g/1$ no puede propagarse de esta manera.

Otra posibilidad consiste en propagar el efecto de la falla a lo largo de dos trayectorias en forma simultánea, como se muestra en la figura 11.4*c*. En este caso la falla se propaga a lo largo

de las trayectorias $g - h - f$ y $g - k - f$. Esto requiere que se establezca $b = 1$ y $c = 1$, lo cual también da la casualidad que es la condición necesaria para causar la propagación como se explicó antes. La prueba 0000 logra el objetivo deseado de detectar $g/1$. Observe que si D (o $\overline{D}$) aparece en ambas entradas de una compuerta NAND, el valor de salida será $\overline{D}$ (o D).

La idea de propagar el efecto de las fallas utilizando la sensibilización de trayectorias se ha aprovechado en varios métodos para la obtención del conjunto de pruebas para la detección de fallas. El esquema ilustrado en la figura 11.4 indica la esencia del algoritmo *D*, que fue uno de los primeros esquemas prácticos desarrollados para propósitos de detección de fallas [7]. Otras técnicas han surgido a partir de este método básico [8].

11.4 CIRCUITOS CON LA ESTRUCTURA DE ÁRBOL

Los circuitos con una estructura tipo árbol, donde cada compuerta tiene una carga de salida *(fan-out)* de 1, son particularmente fáciles de probar. Las formas más comunes de estos circuitos son la suma de productos o el producto de sumas. Como hay una trayectoria única desde cada salida principal a la salida del circuito, basta derivar las pruebas para las fallas en las entradas principales. Ilustraremos este concepto por medio del circuito de suma de productos de la figura 11.5.

Si alguna entrada de una compuerta AND está atascada en 0, esta condición puede detectarse estableciendo todas las entradas de la compuerta en 1 y asegurando que las otras compuertas AND producen 0. Esto hace que $f = 1$ en el circuito en buen estado y $f = 0$ en el circuito defectuoso. Tres de estas pruebas son necesarias debido a que hay tres compuertas AND.

Probar las fallas de atascamiento en 1 es un tanto más complicado. Una entrada de una compuerta AND se prueba para la falla de atascamiento en 1 al aplicarle el valor lógico 0, mientras que las otras entradas de la compuerta tienen el valor lógico 1. Así, una compuerta en buen

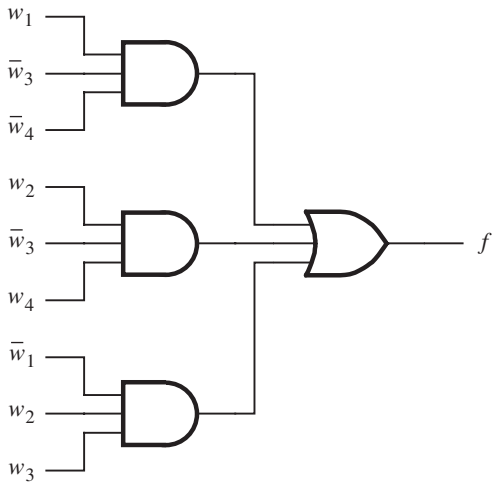


Figura 11.5 Circuito con una estructura de árbol.

	No.	Término producto			Prueba
		$w_1\overline{w}_3\overline{w}_4$	$w_2\overline{w}_3w_4$	$\overline{w}_1w_2w_3$	$w_1w_2w_3w_4$
Pruebas de atascamiento en 0	1	1 1 1	0 1 0	0 0 0	1 0 0 0
	2	0 1 0	1 1 1	1 1 0	0 1 0 1
	3	0 0 0	1 0 1	1 1 1	0 1 1 1
Pruebas de atascamiento en 1	4	0 1 1	1 1 0	1 1 0	0 1 0 0
	5	1 0 1	1 0 0	0 1 1	1 1 1 0
	6	1 1 0	0 1 1	0 0 0	1 0 0 1
	7	1 0 0	1 0 1	0 1 1	1 1 1 1
	8	0 0 0	0 0 1	1 0 1	0 0 1 1

Figura 11.6 Obtención de pruebas para el circuito de la figura 11.5.

estado produce la salida 0 y una compuerta defectuosa genera 1. Al mismo tiempo, las demás compuertas AND deben tener la salida 0, lo cual se logra haciendo que cuando menos una entrada de estas compuertas sea igual a 0.

En la figura 11.6 se muestra la derivación de las pruebas necesarias. Las primeras tres pruebas son para las fallas de atascamiento en 0. La cuarta detecta una falla de atascamiento en 1 ya sea en la primera entrada de la compuerta AND superior o en las terceras entradas de las otras dos compuertas. Obsérvese que en cada caso la entrada probada es manejada por el 0 lógico, mientras que las otras entradas son iguales a 1. Esto genera el vector de prueba $w_1w_2w_3w_4 = 0100$. Desde luego, es útil probar varias entradas en cuantas compuertas sea posible utilizando un solo vector de prueba. La quinta prueba detecta una falla ya sea en la segunda entrada de la compuerta superior o en la primera entrada de la compuerta inferior; no prueba ninguna entrada de la compuerta media. El patrón de pruebas necesario es 1110. Se requieren tres pruebas más para detectar las fallas de atascamiento en las entradas restantes de las compuertas AND. Por consiguiente, el conjunto de pruebas completo es

$$\text{Conjunto de pruebas} = \{1000, 0101, 0111, 0100, 1110, 1001, 1111, 0011\}$$

11.5 PRUEBAS ALEATORIAS

Hasta ahora hemos considerado la tarea de derivar un conjunto de pruebas *determinístico* para un circuito, basándonos principalmente en el concepto de sensibilización de trayectorias. En general, es difícil generar estos conjuntos de pruebas cuando los circuitos son más grandes. Una alternativa útil es elegir las pruebas aleatorias, tema que exploraremos en esta sección.

En la figura 11.7 se proporcionan todas las funciones de dos variables. Para una función de n variables, existen 2^{2^n} funciones posibles; por consiguiente existen $2^{2^2} = 16$ funciones de dos variables. Considérese la función XOR, implementada como se muestra en la figura 11.8. Pensemos en las posibles fallas de atascamiento en 0 y atascamiento en 1 en los cables *b, c, d, h* y

w_1w_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figura 11.7 Todas las funciones de dos variables.

k de este circuito. Cada falla transforma el circuito en un circuito defectuoso que implementa una función distinta de XOR, como se indica en la figura 11.9. Para probar el circuito podemos aplicar una o más combinaciones de entrada a fin de distinguir el circuito en buen estado de los posibles circuitos defectuosos enumerados en la figura 11.9. Elija arbitrariamente $w_1w_2 = 01$ como la primera prueba, con la que distinguirá el circuito en buen estado, que debe generar $f = 1$, de los circuitos defectuosos, que producen f_0, f_2, f_3 y f_{10}, ya que cada uno de estos generará $f = 0$. A continuación elija en forma arbitraria la prueba $w_1w_2 = 11$. Esta prueba distingue el circuito en buen estado de los defectuosos que producen f_5, f_7 y f_{15}, además de f_3, el cual ya probamos mediante $w_1w_2 = 01$. Supóngase que la tercera prueba es $w_1w_2 = 10$; diferenciará el circuito en buen estado de f_4 y f_{12}. Estas tres pruebas, elegidas aparentemente al azar, detectan todos los circuitos defectuosos relacionados con las fallas de la figura 11.9. Asimismo, nótese que las primeras dos pruebas distinguen siete de los nueve circuitos defectuosos posibles.

Este ejemplo sugiere que es posible obtener un conjunto de pruebas adecuado al seleccionar las pruebas de manera aleatoria. ¿Qué tan eficaces pueden ser las pruebas aleatorias? Al observar la figura 11.7 se advierte que cualquiera de las cuatro pruebas posibles distingue la función correcta de ocho funciones con errores, pues producen diferentes valores de salida para esta combinación de entrada. Estas ocho funciones detectables mediante una prueba individual son la mitad del número total de funciones posibles (2^{2^2-1} para el caso de las dos variables). La prueba no puede diferenciar entre la función correcta y las siete funciones con fallas que producen el mismo valor de salida. La aplicación de la segunda prueba distingue cuatro de las siete funciones

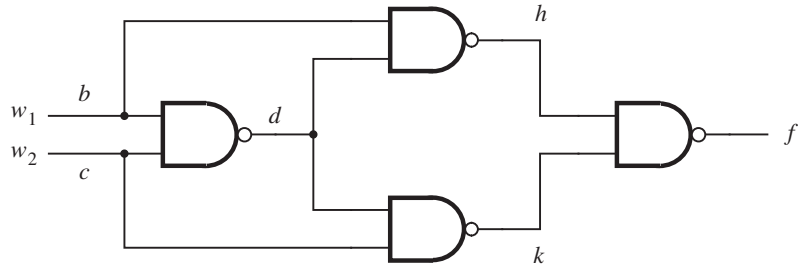


Figura 11.8 El circuito XOR.

Falla	Implementaciones del circuito
$b/0$	$f_5 = w_2$
$b/1$	$f_{10} = \overline{w}_2$
$c/0$	$f_3 = w_1$
$c/1$	$f_{12} = \overline{w}_1$
$d/0$	$f_0 = 0$
$d/1$	$f_7 = w_1 + w_2$
$h/0$	$f_{15} = 1$
$h/1$	$f_4 = \overline{w}_1 w_2$
$k/0$	$f_{15} = 1$
$k/1$	$f_2 = w_1 \overline{w}_2$

Figura 11.9 El efecto de varias fallas.

restantes debido a que producen un valor de salida diferente de la función correcta. Por tanto, cada aplicación de una prueba nueva esencialmente reduce a la mitad el número de funciones con fallas que no se han detectado. En consecuencia, la probabilidad de que las primeras pruebas detectarán una porción grande de todas las fallas posibles es alta. Más específicamente, la probabilidad de que cada circuito defectuoso pueda detectarse es

$$P_1 = \frac{1}{2^{2^2} - 1} \cdot 2^{2^2 - 1} = \frac{8}{15} = 0.53$$

Ésta es la proporción del número de circuitos defectuosos que producen un valor de salida diferente de aquel del circuito en buen estado para el número total de circuitos defectuosos.

Este razonamiento se extiende sin problemas a las funciones de n variables. En este caso la primera prueba detecta $2^{2^n - 1}$ de un total de $2^{2^n} - 1$ posibles funciones con fallas. Por tanto, si se aplican m pruebas, la probabilidad de que un circuito con fallas sea detectado es

$$P_m = \frac{1}{2^{2^n} - 1} \cdot \sum_{i=1}^{m} 2^{2^n - i}$$

Esta expresión se representa en forma gráfica en la figura 11.10. La conclusión es que las pruebas aleatorias son muy eficaces y que después de unas cuantas decenas de pruebas es probable que la existencia de una falla se detecte incluso en circuitos muy grandes.

Las pruebas aleatorias funcionan particularmente bien para circuitos que no tienen una carga de entrada grande. Si la carga de entrada es grande, tal vez sea necesario recurrir a otros esquemas de pruebas. Supóngase que una compuerta AND tiene un número grande de entradas. Entonces hay un problema con la detección de las fallas de atascamiento en 1, las cuales tal vez no estén cubiertas por pruebas aleatorias. Pero es posible probar estas fallas utilizando el método descrito en la sección 11.4.

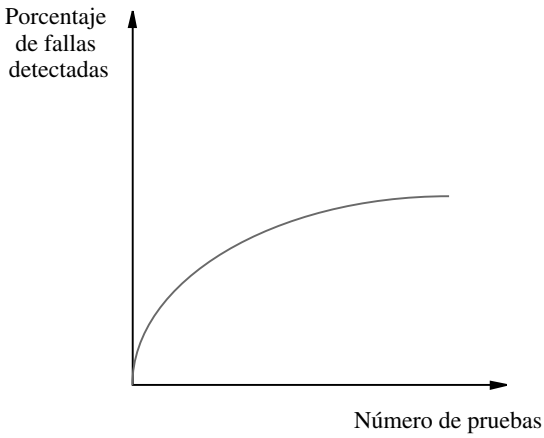


Figura 11.10 Eficacia de las pruebas aleatorias.

La simplicidad de las pruebas aleatorias es una característica muy atractiva. Por ello, junto con la buena eficacia de las pruebas, esta técnica se utiliza mucho en la práctica.

11.6 PRUEBAS DE CIRCUITOS SECUENCIALES

Como vimos en las secciones anteriores, los circuitos combinacionales pueden probarse de manera eficaz usando conjuntos de pruebas determinísticos o aleatorios. Es mucho más difícil probar los circuitos secuenciales. La presencia de elementos de memoria permite que un circuito secuencial se halle en varios estados, y la respuesta del circuito a las entradas de prueba aplicadas externamente depende del estado del circuito.

Un circuito combinacional puede probarse comparando su comportamiento con la funcionalidad especificada en la tabla de verdad. Un intento equivalente sería probar un circuito secuencial comparando su comportamiento con la funcionalidad especificada en la tabla de estado. Esto supone revisar que el circuito realiza correctamente todas las transiciones entre los estados y que produce una salida correcta. Este enfoque puede parecer sencillo, pero en realidad es muy difícil. Un gran obstáculo es la dificultad para determinar que el circuito se encuentra en un estado específico si las variables de estado no son observables en los pines externos del circuito, lo que, en general, es el caso. Aún así, por cada transición que va a probarse es necesario verificar con toda certeza que se llegó al estado de destino correcto. Un enfoque como éste puede funcionar bien para circuitos secuenciales muy pequeños, pero no es factible para circuitos de tamaño real. Un método mucho mejor es diseñar los circuitos secuenciales de tal manera que sea fácil aplicarles las pruebas.

11.6.1 DISEÑO PARA LA APLICACIÓN DE PRUEBAS

Un circuito secuencial síncrono comprende el circuito combinacional que implementa la salida y las funciones de estado siguiente, así como los flip-flops que almacenan la información del estado durante un ciclo de reloj. Un modelo general para los circuitos secuenciales se muestra

en la figura 8.90. Las entradas a la red combinacional son las entradas primarias, de w_1 a w_n, y las variables de estado actual, de y_1 a y_k. Las salidas de la red son las salidas primarias, de z_1 a z_m, y las variables de estado siguiente son de Y_1 a Y_k. La red combinacional podría probarse con las técnicas presentadas en secciones anteriores si fuera posible aplicar pruebas en todas sus entradas y observar los resultados de todas sus salidas. La aplicación de vectores de prueba a las entradas primarias no representa ninguna dificultad. Además, es fácil observar los valores en las salidas primarias. La pregunta es cómo aplicar los vectores de prueba en las entradas del valor actual y cómo observar los valores en las salidas del estado siguiente.

Un enfoque posible consiste en incluir un multiplexor de dos vías en la trayectoria de cada variable de estado actual de modo que la entrada a la red combinacional pueda ser ya sea el valor de la variable de estado (obtenida desde la salida del flip-flop correspondiente) o el valor que sea una parte del vector de prueba. Una desventaja importante de este enfoque es que la segunda entrada de cada multiplexor debe estar accesible directamente a través de los pines externos, lo cual requiere muchos pines si existen muchas variables de estado. Una alternativa más atractiva es proporcionar una conexión que permita desplazar el vector de prueba en el circuito un bit a la vez, compensando así los requisitos del pin durante el tiempo que se precisa realizar una prueba. Se han propuesto varios esquemas como éste, uno de los cuales se describe enseguida.

Técnica de exploración de trayectorias

Una técnica popular, llamada *exploración de trayectorias,* utiliza multiplexores en las entradas de los flip-flops para permitir que éstos se usen en forma independiente durante la operación normal del circuito secuencial o como parte de un registro de corrimiento para propósitos de pruebas. En la figura 11.11 se presenta la estructura general de exploración de trayectoria para un circuito con tres flip-flops. Un multiplexor dos a uno conecta la entrada D de cada flip-flop a la variable de estado siguiente respectiva o a la trayectoria serial que conecta todos los flip-flops en un registro de corrimiento. La señal de control $\overline{Normal}/Scan$ selecciona la entrada activa del multiplexor. Durante la operación normal las entradas del flip-flop son manejadas por las variables de estado siguiente, Y_1, Y_2 y Y_3.

Para propósitos de prueba la conexión del registro de corrimiento se usa a fin de explorar la parte de cada vector de pruebas que comprende las variables de estado actual, y_1, y_2 y y_3. Esta conexión tiene Q_i conectada a D_{i+1}. La entrada al primer flip-flop es el pin accesible externamente *Scan-in.* La salida proviene del último flip-flop, el cual se proporciona en el pin *Scan-out*.

La técnica de exploración de trayectoria comprende los pasos siguientes:

1. La operación de los flip-flops se prueba explorándolos en un patrón de 0 y 1, por ejemplo, 01011001, en ciclos de reloj consecutivos, y observando si se obtiene el mismo patrón de salida.
2. El circuito combinacional se prueba aplicando vectores de prueba a $w_1w_2 \cdots w_n y_1 y_2 y_3$ y observando los valores generados por $z_1z_2 \cdots z_m Y_1Y_2Y_3$. Esto se hace como sigue:
   - La porción $y_1 y_2 y_3$ del vector de prueba se explora en los flip-flops durante tres ciclos de reloj usando $\overline{Normal}/Scan = 1$.
   - La parte $w_1w_2 \cdots w_n$ del vector de prueba se aplica como de costumbre y la operación normal del circuito secuencial se realiza durante un ciclo de reloj, al establecer $\overline{Normal}/Scan = 0$. Se observan las salidas $z_1z_2 \cdots z_m$. Los valores generados de $Y_1Y_2Y_3$ se cargan en los flip-flops en este momento.
   - La entrada select se cambia a $\overline{Normal}/Scan = 1$, y el contenido de los flip-flops se explora durante los tres ciclos de reloj siguientes, lo que permite que la parte $Y_1Y_2Y_3$ del

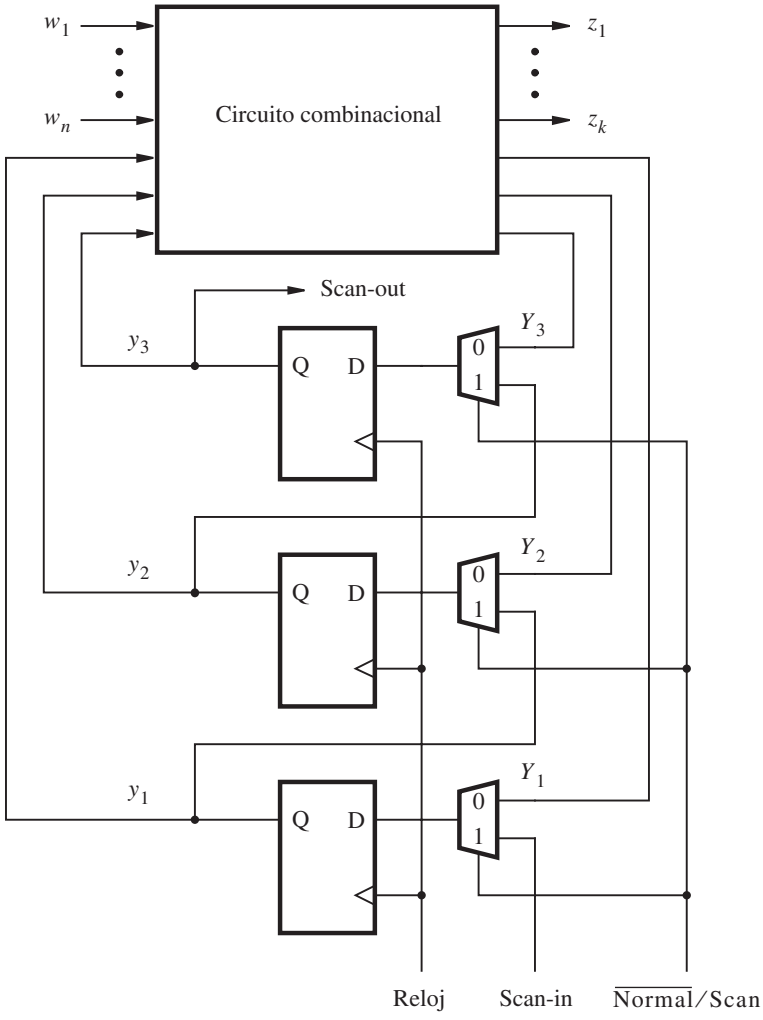


Figura 11.11 Arreglo de exploración de trayectoria.

resultado de la prueba pueda observarse externamente. Al mismo tiempo, el siguiente vector de prueba puede explorarse para reducir el tiempo total requerido para probar el circuito.

En el ejemplo siguiente se muestra un circuito específico que está diseñado para las pruebas de exploración de trayectoria.

Ejemplo 11.3

En la figura 8.80 se ilustra un circuito que reconoce una secuencia de entrada específica, la cual se estudió en la sección 8.9. Es fácil aplicar las pruebas al circuito modificándolo para la exploración de trayectorias como se indica en la figura 11.12. La parte combinacional, que consta de cuatro compuertas AND y dos OR, es la misma en ambas figuras.

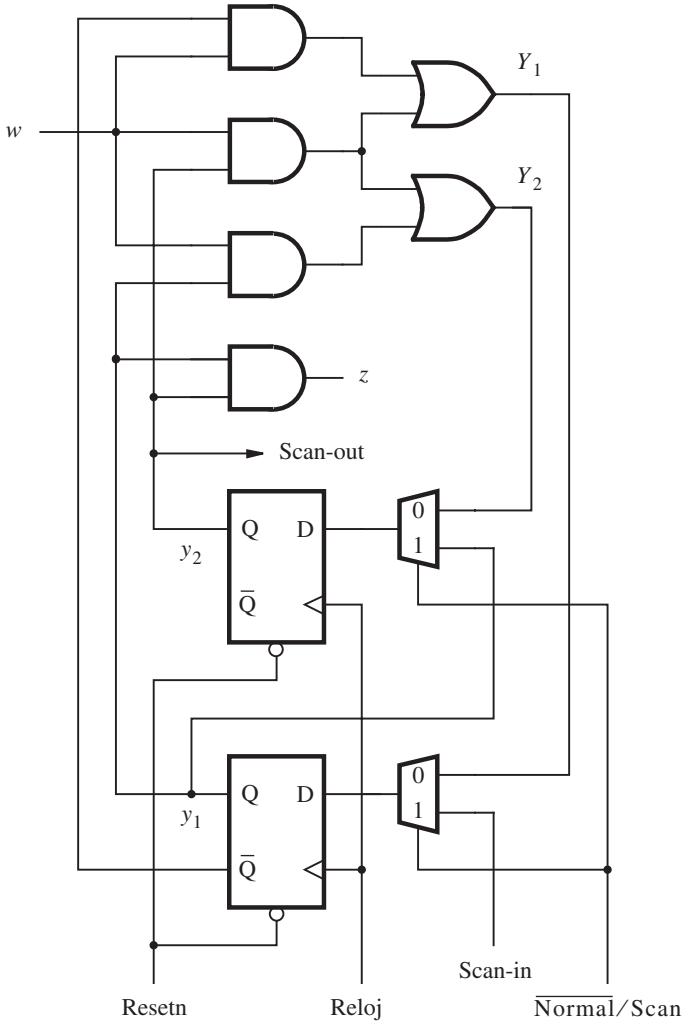


Figura 11.12 Circuito para el ejemplo 11.3.

Los flip-flops pueden probarse explorando a través de ellos una secuencia de 0 y 1 como se explicó líneas arriba. El circuito combinacional puede probarse aplicando vectores de prueba en w, y_1 y y_2. Usemos el método de pruebas aleatorias, eligiendo en forma arbitraria cuatro vectores de prueba wy_1y_2 = 001, 110, 100 y 111. Para aplicar el primer vector de prueba, el patrón y_1y_2 = 01 se explora en los flip-flops durante dos ciclos de reloj. Entonces para un ciclo de reloj se hace que el circuito opere en el modo normal con w = 0. Esto aplica en esencia el vector wy_1y_2 = 001 al circuito AND-OR. El resultado de esta prueba debe ser z = 0, Y_1 = 0 y Y_2 = 0. El valor de z puede observarse directamente. Los valores de Y_1 y Y_2 se cargan en los flip-flops respectivos, los cuales se exploran durante los dos ciclos siguientes de reloj. A medida que estos valores se revisan, el patrón de pruebas y_1y_2 = 10 siguiente puede explorarse. Por tanto, realizar una

prueba toma cinco ciclos, pero los dos últimos se traslapan con la segunda prueba. La tercera y cuarta pruebas se llevan a cabo de la misma forma. El tiempo total requerido para realizar las cuatro pruebas es 14 ciclos de reloj.

El enfoque anterior se basa en la aplicación de pruebas a un circuito secuencial probando su parte combinacional por medio de las técnicas expuestas en las secciones anteriores. La facilidad de exploración de trayectorias también vuelve posible probar el circuito secuencial haciendo que pase por todas las transiciones especificadas en la tabla de estado. El circuito puede colocarse en cierto estado simplemente explorando en los flip-flops la combinación de las variables de estado que denotan ese estado. El resultado de la transición puede revisarse observando las salidas primarias y explorando la combinación que presenta el estado de destino. Dejamos al lector el desarrollo de los detalles de este método (véase el problema 11.16).

Una limitación de la técnica de exploración de trayectoria es que no trabaja bien si las funciones *preset* y *reset* asíncronas de los flip-flops se usan durante la operación normal. Ya hemos sugerido que es mejor utilizar *preset* y *reset* síncronos. Si el diseñador quiere emplear la capacidad de *preset* y *reset* asíncronos, entonces un circuito al que se aplican las pruebas puede diseñarse mediante técnicas como la del *diseño de exploración sensible al nivel*. El lector puede consultar el material de referencia [1, 9] para obtener una descripción de esta técnica.

11.7 PRUEBA AUTOMATIZADA INTEGRADA

Hasta ahora hemos supuesto que las pruebas a los circuitos lógicos se realizan aplicando externamente las entradas de prueba y comparando los resultados con el comportamiento esperado del circuito. Ello requiere conectar el equipo externo al circuito bajo prueba. Una pregunta interesante es si es posible incorporar la capacidad de aplicación de pruebas dentro del circuito de modo que no se necesite ningún equipo externo. Esta capacidad integrada permitirá al circuito la aplicación automática de pruebas. En esta sección presentamos un esquema que brinda la capacidad de pruebas automáticas integradas (BIST, *built-in self-test*).

En la figura 11.13 se muestra un posible arreglo BIST en el que un generador de vectores de prueba produce los que deben aplicarse al circuito que se analiza. En la sección 11.5 explicamos que los vectores de prueba elegidos al azar dan buenos resultados, y que la cobertura de fallas depende del número de pruebas realizadas. Por cada vector de prueba aplicado al circuito, es ne-

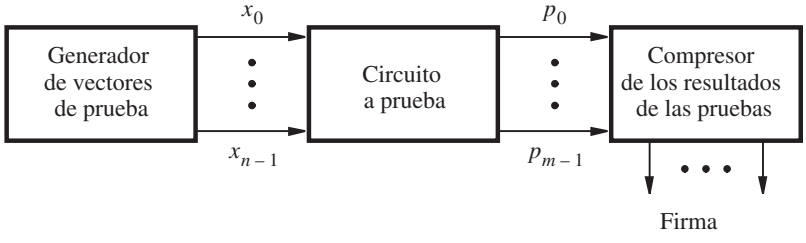
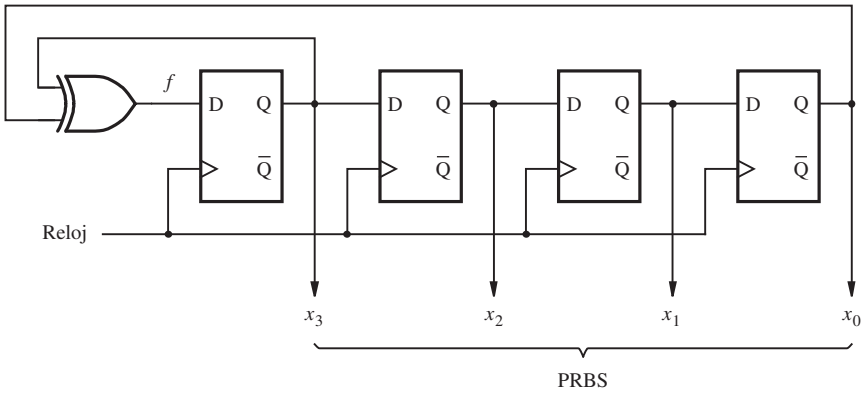


Figura 11.13 El arreglo para pruebas.

cesario determinar la respuesta requerida del circuito. La respuesta de un circuito en buen estado puede determinarse utilizando la herramienta de simulación de un sistema CAD. Las respuestas esperadas para las pruebas aplicadas deben almacenarse en el chip de modo que pueda hacerse una comparación cuando el circuito se esté probando.

Un enfoque práctico para generar los vectores de prueba integrados a los chips consiste en utilizar pruebas *seudoaleatorias*, las cuales tienen las mismas características que las pruebas aleatorias pero se producen en forma determinística y pueden repetirse a voluntad. El generador de pruebas seudoaleatorias se construye fácilmente ocupando un circuito de registros de corrimiento de retroalimentación. En la figura 11.14 se presenta un pequeño ejemplo de un generador posible. Un registro de corrimiento de cuatro bits, con las señales de la primera y cuarta etapas retroalimentadas a través de una compuerta XOR, genera 15 patrones durante ciclos de reloj sucesivos. Si el registro de corrimiento se establece al principio en $x_3x_2x_1x_0 = 1000$, entonces los patrones generados se muestran en el inciso *(b)* de la figura. Observe que el patrón 0000 no puede usarse, ya que el circuito estaría atrapado en él de forma indefinida.

El circuito de la figura 11.14 es representativo de un tipo de circuitos conocido como *registros de corrimiento con retroalimentación lineal* (LFSR, *linear feedback shift registers*). Al usar la



a) Circuito

x_3	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	···
x_2	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	···
x_1	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	···
x_0	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	···
f	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	1	···

b) Secuencia generada

Figura 11.14 Generador seudoaleatorio de secuencias binarias (PRBSG, *pseudorandom binary sequence generator*).

retroalimentación desde las diversas etapas de un registro de corrimiento de n bits conectado a la primera etapa por medio de compuertas XOR, es posible generar una secuencia de $2^n - 1$ patrones que tengan las características de los números generados al azar. Estos circuitos se usan exhaustivamente en los códigos de corrección de errores. La teoría de su operación se expone en varios libros [1-3, 10]. Peterson y Weldon [11] ofrecen una tabla de las conexiones de retroalimentación para varios valores de n, la cual genera una secuencia seudoaleatoria de longitud máxima.

El generador de secuencias binarias seudoaleatorias (PRBSG, *pseudorandom binary sequence generator*) provee un método simple de generación de pruebas. La respuesta requerida del circuito al que se aplican las pruebas puede determinarse usando la herramienta simuladora del sistema CAD. La pregunta que falta por responder es cómo verificar si el circuito produce la respuesta requerida. No es atractivo tener que almacenar un número grande de respuestas a las pruebas en un chip que además incluye el circuito principal. Una solución práctica consiste en comprimir los resultados de las pruebas en un solo patrón. Esto puede hacerse con un circuito LFSR. En vez de proporcionar únicamente las señales de retroalimentación como entrada, un circuito compresor incluye las señales de salida producidas por el circuito que se prueba. En la figura 11.15 se muestra un circuito compresor de una sola entrada (SIC, *single-input compressor*), que usa las mismas conexiones de retroalimentación que el PRBSG de la figura 11.14. La entrada p es la salida de un circuito bajo prueba. Después de aplicar una serie de vectores de prueba, los valores resultantes de p manejan el SIC y, junto con la funcionalidad LFSR, producen un patrón de cuatro bits. El patrón generado por el SIC se llama *firma* del circuito probado para la secuencia de pruebas específica. La firma representa un solo patrón que puede interpretarse como resultado de todas las pruebas aplicadas. Puede compararse contra un patrón predeterminado para ver si el circuito probado trabaja bien. El almacenamiento de un patrón individual de n bits para propósitos de comparación presenta sólo un pequeño inconveniente. La naturaleza aleatoria de los circuitos compresores basados en los LFSR ofrece una buena cobertura de los patrones que pueden resultar de un circuito defectuoso [12].

Si el circuito que se prueba tiene más de una salida, entonces puede usarse un LSFR con varias entradas. En la figura 11.16 se ilustra cómo cuatro entradas, de p_0 a p_3, pueden añadirse al circuito básico de la figura 11.14. De nuevo la firma de cuatro bits proporciona un buen mecanismo para distinguir entre las diferentes secuencias de patrones de cuatro bits que pueden aparecer en las entradas de este circuito compresor de varias entradas (MIC, *multiple-input compressor circuit*).

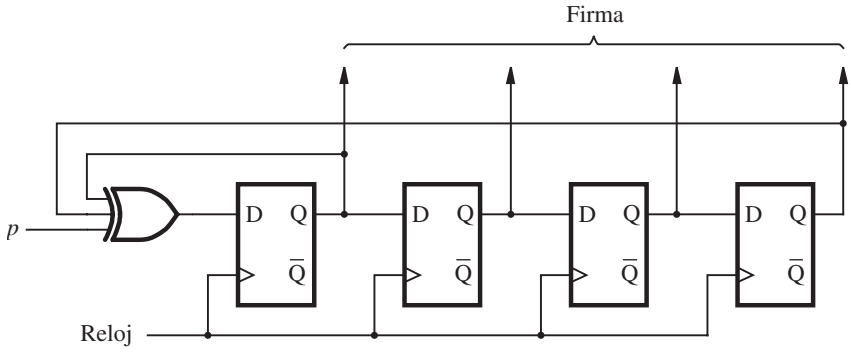


Figura 11.15 Circuito compresor de una sola entrada (SIC).

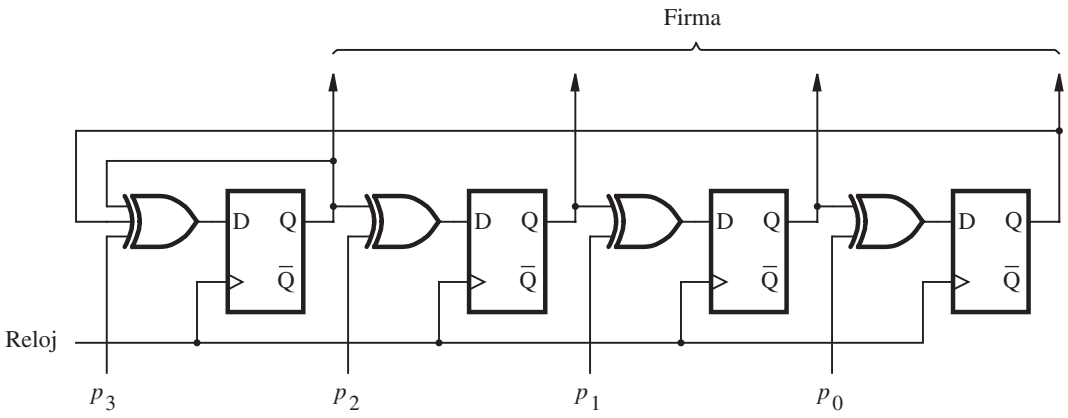


Figura 11.16 Circuito compresor de varias entradas (MIC).

Un esquema BIST completo para un circuito secuencial puede implementarse como se indica en la figura 11.17. El método de exploración de trayectoria se usa para proporcionar un circuito al que pueden aplicarse pruebas. Los patrones de prueba que normalmente se aplicarían a las entradas principales $W = w_1 w_2 \cdots w_n$ se generan internamente como los patrones en $X = x_1 x_2 \cdots x_n$. Se necesitan los multiplexores para permitir el intercambio de W a X como entradas al circuito combinacional. Un generador de secuencias binarias seudoaleatorias, PRBSG-X, genera los

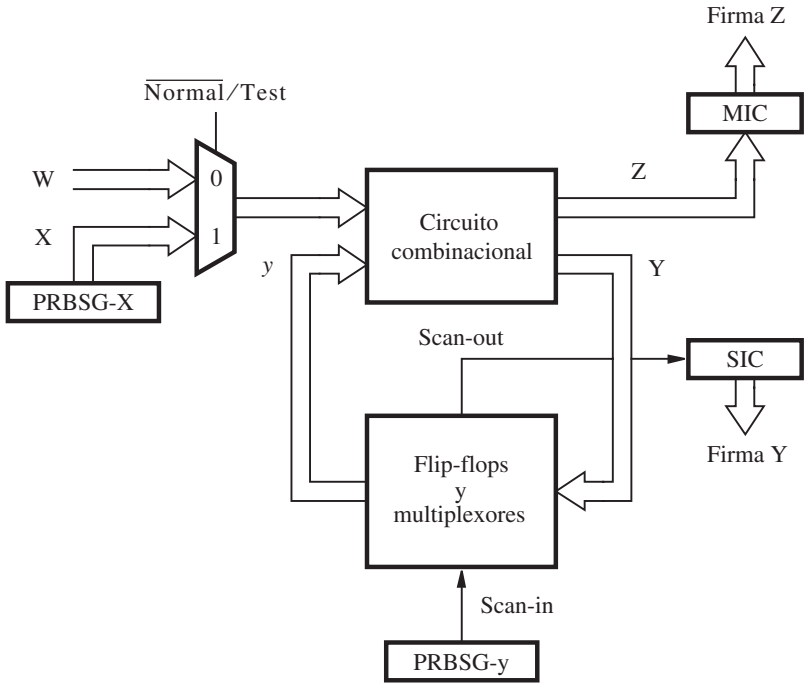


Figura 11.17 BIST en un circuito secuencial.

patrones de prueba para *X.* La parte de las pruebas aplicadas por medio de las señales de estado siguiente, *y,* es generada por el segundo generador PRBS, PRBSG-y. Estos patrones se exploran en los flip-flops como se explicó en la sección 11.6.

Las salidas de prueba se comprimen utilizando los dos circuitos compresores. Los patrones en las salidas principales, $Z = z_1 z_2 \cdots z_m$, se comprimen usando el circuito MIC, y aquellos en los cables de estado siguiente $Y = Y_1 Y_2 \cdots Y_k$, por el circuito SIC. Estos circuitos producen la firma *Z* y la firma *Y*, respectivamente. El procedimiento de prueba es el mismo que el del ejemplo 11.3, excepto porque la comparación con el resultado de prueba que se supone da un buen circuito se realiza sólo una vez; al final del proceso de prueba las dos firmas se comparan con los patrones almacenados. En la figura 11.17 no se muestra el sistema de circuitos necesario para almacenar estos patrones y realizar la comparación. En vez de almacenar los patrones de firma de los resultados requeridos como parte del circuito diseñado, es posible desplazar el contenido de los registros de corrimiento de MIC y SIC en dos pines de salida y efectuar la comparación necesaria con las firmas esperadas externamente. Nótese que al aplicar la prueba de la firma de esta manera se reduce el tiempo considerablemente en comparación con el tiempo que llevaría probar el circuito explorando los resultados de pruebas individuales y comparándolos con los patrones predeterminados.

La efectividad del método BIST depende de la longitud del generador LFSR y los circuitos compresores. Los registros de corrimiento más grandes dan mejores resultados [13]. Una razón para no detectar que el circuito bajo prueba puede tener fallas es que los exámenes generados seudoaleatoriamente no tienen una cobertura perfecta de todas las fallas posibles. Otra razón es que una firma generada mediante la compresión de las salidas de un circuito defectuoso puede terminar siendo por casualidad la misma firma generada por el circuito en buen estado. Esto puede ocurrir debido a que el proceso de compresión resulta en la pérdida de cierta información, como el hecho de que dos patrones de salida distintos pueden comprimirse en la misma firma. Este inconveniente se conoce como *solapamiento* (*aliasing*).

11.7.1 OBSERVADOR DE BLOQUES LÓGICOS INTEGRADO

La esencia de BIST es tener la capacidad interna para la generación de pruebas y para la compresión de los resultados. En vez de utilizar circuitos separados para estas dos funciones, es posible diseñar un solo circuito que sirva para ambos propósitos. En la figura 11.18 se muestra la estructura de un circuito posible, conocido como *observador de bloques lógicos integrado* (BILBO, *built-in logic block observer*) [14]. Este circuito de cuatro bits tiene las mismas conexiones de retroalimentación que el de la figura 11.14.

El circuito BILBO tiene cuatro modos de operación, los cuales están controlados por los bits de modo, M_1 y M_2. Los modos son los siguientes:

- $M_1M_2 = 11$ — Modo del sistema normal en el que todos los flip-flops se controlan en forma independiente por medio de señales en las entradas de p_0 a p_3. En este modo cada flip-flop puede usarse para implementar una variable de estado de una máquina de estado finito utilizando p_0 a p_3 como y_0 a y_3.
- $M_1M_2 = 00$ — Modo de registro de corrimiento en el que los flip-flops están conectados a un registro de corrimiento. Este modo permite que se haga una exploración de entrada de los vectores de prueba y una exploración de salida de los resultados de las pruebas aplicadas, si la entrada de control $\overline{G}/S$ es igual a 1. Si $\overline{G}/S = 0$, entonces el circuito actúa como el generador de PRBS.
- $M_1M_2 = 10$ — Modo de firma en el que una serie de patrones aplicados a las entradas p_0 a p_3 se comprimen en una firma disponible como un patrón en q_0 a q_3.
- $M_1M_2 = 01$ — Modo de inicialización en el que todos los flip-flops se inicializan en 0.

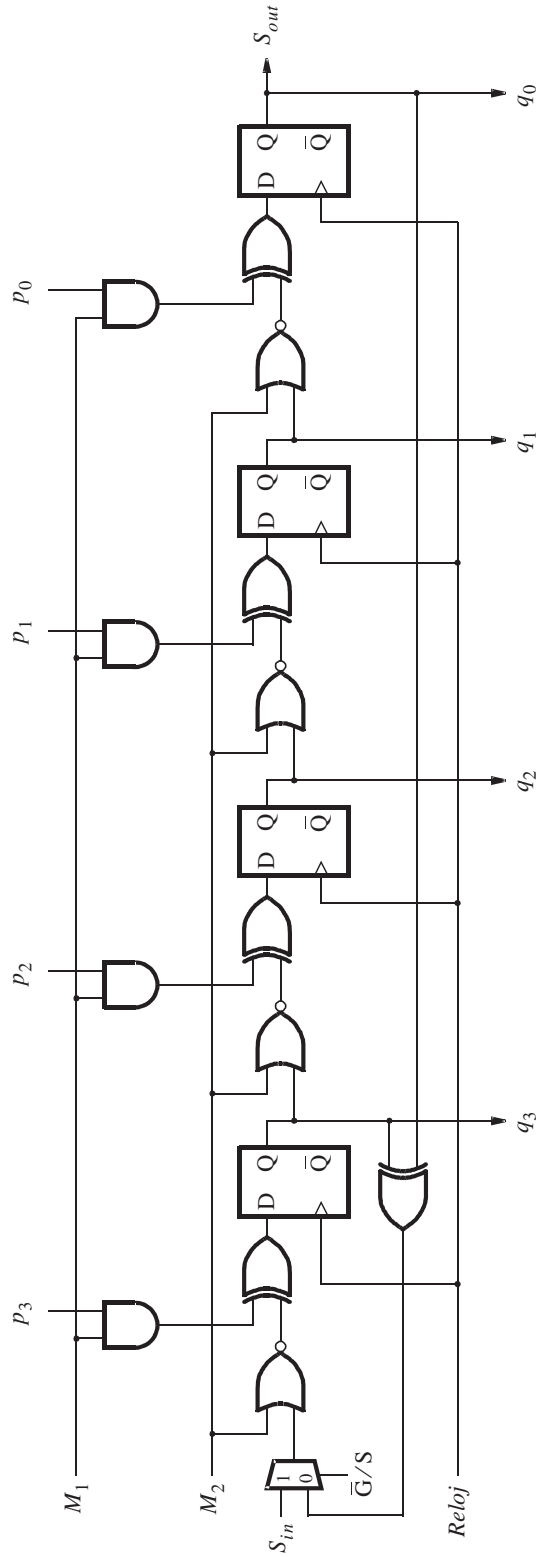


Figura 11.18 Un observador de bloques lógicos integrado (BILBO) de cuatro bits.

Una forma eficaz de utilizar circuitos BILBO se presenta en la figura 11.19. Un circuito combinacional puede probarse partiéndolo en dos (o más) partes. Un circuito BILBO sirve para proporcionar entradas a una parte y aceptar salidas de la otra. El proceso de pruebas supone un método de dos fases. Primero, BILBO1 se utiliza como un generador PRBS que proporciona patrones de prueba para la red combinacional 1 (CN1). Durante este tiempo BILBO2 actúa como un compresor y produce una firma para la prueba. La firma se desplaza hacia fuera colocando BILBO2 en el modo de registro de corrimiento. Enseguida, las funciones de BILBO1 y BILBO2 se invierten y el proceso se repite para la prueba de CN2.

Los pasos detallados del proceso de pruebas son:

1. Explorar el patrón de pruebas inicial hacia BILBO1 e inicializar todos los flip-flops en BILBO2.
2. Usar BILBO1 como el generador PRBS para un número específico de ciclos de reloj y utilizar BILBO2 para producir una firma.
3. Explorar el contenido de BILBO2 y comparar externamente la firma; luego introducir en ella el patrón de pruebas inicial para probar CN2. Inicializar los flip-flops en BILBO1.
4. Usar BILBO2 como el generador de PRBS para un número específico de ciclos de reloj y utilizar BILBO1 para producir una firma.
5. Explorar la firma en BILBO1 y compararla externamente con el patrón requerido.

Los circuitos BILBO se emplean de esta manera con fines de prueba. Otras veces se utiliza el modo de sistema normal.

11.7.2 ANÁLISIS DE FIRMAS

Hemos explicado el uso de firmas en el contexto de la implementación de un mecanismo de pruebas integrado que sea eficaz. La idea central de comprimir una secuencia larga de pruebas que dan como resultado una sola firma originalmente se desarrolló como la base para un instrumento fabricado por Hewlett-Packard en la década de 1970, conocido como *analizador de firmas* (*signature analyzer*) [15]. Por tanto, el nombre *análisis de firmas* se acuñó para referirse a los esquemas de prueba que utilizan firmas para representar los resultados de las pruebas aplicadas.

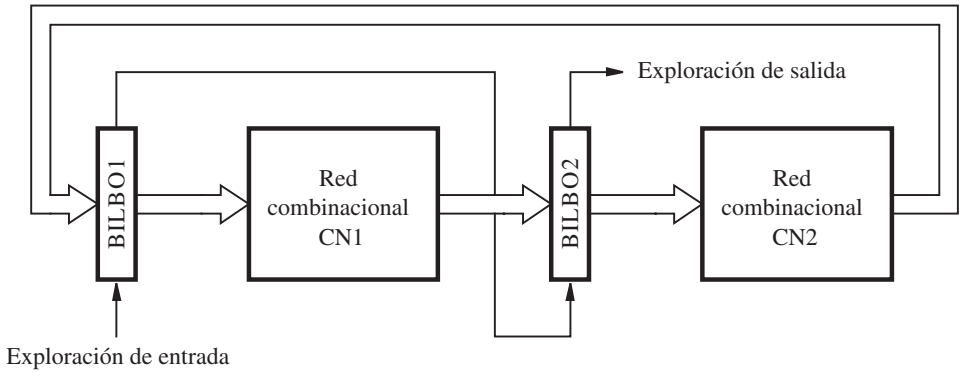


Figura 11.19 Uso de circuitos BILBO para pruebas.

El análisis de firmas es particularmente adecuado para los sistemas digitales que incluyen por naturaleza la capacidad de generar los patrones de prueba deseados. Tal es el caso de los sistemas de cómputo en los que varias partes del sistema pueden estimularse por los patrones de prueba producidos bajo el control de software.

11.7.3 BOUNDARY SCAN

Las técnicas de pruebas expuestas en las secciones anteriores se aplican por igual a los circuitos que se implementan en chips individuales o en tarjetas de circuitos impresos que contienen una serie de chips. Un circuito puede probarse sólo si es posible aplicarle las pruebas y observar las salidas producidas. Esto implica tener acceso a las entradas y salidas principales.

Cuando los chips están soldados en una tarjeta de circuito impreso se vuelve casi imposible conectar sondas de prueba a los pines. Esto dificulta el proceso de pruebas a menos que se proporcione algún acceso indirecto a los pines. El concepto de exploración de trayectoria puede ampliarse al nivel de la tarjeta para abordar el problema. Supóngase que cada pin de entrada o salida principal en un chip está conectado por medio de un flip-flop D y que se prevé un modo de prueba en el que todos los flip-flops pueden conectarse en un registro de corrimiento. Entonces la información de la prueba puede explorarse en la entrada y la salida utilizando la trayectoria del registro de corrimiento, a través de dos pines que sirven como entrada y salida seriales. Conectar el pin de salida serial de un chip al pin de entrada serial de otro da como resultado que los pines de todos los chips se conecten a un registro de corrimiento del ancho de la tarjeta para propósitos de prueba. Este enfoque se ha vuelto popular en la práctica y se ha incorporado en la norma 1149.1 del IEEE [16].

11.8 TARJETAS DE CIRCUITOS IMPRESOS

Las técnicas de diseño y pruebas presentadas en esta obra pueden aplicarse a cualquier circuito lógico, sin importar si se produce en un solo chip o su implementación implica una serie de chips colocados en una tarjeta de circuito impreso (PCB, *printed circuit board*). En esta sección estudiamos algunos problemas prácticos que surgen cuando uno o más circuitos que forman un sistema digital más grande se implementan en una PCB.

Una PCB típica contiene varias capas de cables. Cuando se fabrica la tarjeta se genera el patrón de cables de cada capa. Las capas están separadas por un material aislante y se prensan juntas a manera de sándwich para formar la tarjeta. Las conexiones entre los diferentes niveles de cables se hacen por medio de agujeros provistos para este fin. Los chips y otros componentes luego se sueldan a la parte superior y posiblemente a las capas inferiores.

En capítulos anteriores hemos visto con gran detalle las herramientas utilizadas para el diseño de circuitos que pueden implementarse en un solo chip, digamos un PLD. Para una implementación de varios chips necesitamos un conjunto diferente de herramientas CAD para diseñar una PCB que incorpore los chips y las conexiones necesarias para producir el sistema digital completo. Hay varias herramientas de éstas procedentes de diversas compañías, entre ellas Cadence Design Systems y Mentor Graphics. Estas herramientas pueden determinar automáticamente dónde debe colocarse cada chip en la PCB, pero el diseñador también puede especificar la ubicación de los chips. A esto se le llama *proceso de colocación*. Dada una colocación específica de chips y otros componentes (como conectores y capacitores), las herramientas generan una plantilla para cada capa de señales de cables que proporcionan las conexiones requeridas en la

tarjeta. Este proceso se denomina *enrutamiento*. De nuevo, el diseñador puede intervenir y enrutar manualmente algunas conexiones. Sin embargo, puesto que el número de conexiones puede rondar las decenas de miles, es indispensable obtener una buena solución automatizada.

Además de los aspectos de diseño estudiados en capítulos anteriores, un circuito grande implementado en una PCB está sujeto a otras restricciones. Las señales en los cables pueden verse afectadas por problemas de ruido causados por interferencia, picos en el voltaje y reflexiones de los puntos finales de las rutas largas.

Interferencia

Dos cables colocados muy cercanos entre sí que corren en paralelo están acoplados capacitivamente, y un pulso en un cable puede inducir un pulso similar (pero en general mucho más pequeño) en el cable contiguo. Esto se conoce como *interferencia.* Su existencia es indeseable porque contribuye a ocasionar problemas de ruido.

Cuando se trazan diagramas de tiempo, casi siempre se dibujan formas de ondas ideales con bordes nítidos, que tienen bien definidos los niveles de voltaje para los valores lógicos 0 y 1. En un circuito real las señales correspondientes pueden apartarse significativamente del comportamiento deseado. Como explicamos en la sección 3.8.4, el ruido en un circuito puede afectar los niveles de voltaje, lo que quizá sea molesto. Por ejemplo, si en algún punto en el tiempo el ruido disminuye el valor de una señal que ha de estar en un 1 lógico a un nivel donde esta señal se interpreta por la compuerta siguiente como un 0 lógico, entonces es probable que el circuito funcione mal. Como los efectos del ruido tienden a ser aleatorios, suele ser difícil detectarlos.

Para reducir al mínimo la interferencia es prudente evitar tener cables largos que corran paralelos muy cerca unos de otros, lo cual puede ser difícil de lograr debido al espacio limitado en una PCB y la necesidad de proporcionar un número grande de cables. El uso de capas (planos) adicionales de cables ayuda a enfrentar los problemas de interferencia.

Ruido en la fuente de alimentación

Cuando un circuito CMOS cambia su estado hay un flujo de corriente momentáneo en el circuito, que se manifiesta como un pulso de corriente en los cables de alimentación de energía *(V_{DD}* y *Ground)*. Como una ruta de alambrado en una PCB tiene una pequeña "inductancia de línea", un pulso de corriente como ése causa un pico de voltaje (pulso corto) en esas líneas. El efecto acumulativo de estos picos de voltaje puede ocasionar un malfuncionamiento del circuito.

Los picos de voltaje inducidos pueden reducirse considerablemente si se conecta un capacitor pequeño entre los cables V_{DD} y *Ground*, en estrecha proximidad con el chip que hace que los picos ocurran. Puesto que estos picos tienen la característica de una señal de frecuencia muy alta, la trayectoria a través del capacitor es, en esencia, un cortocircuito para ellas. Por tanto, los picos de voltaje "pasan por (*bypass*)" las líneas de alimentación de energía y no afectan otros chips conectados a ellas. Estos capacitores se llaman *capacitores de desacoplamiento (bypass capacitors*). No afectan el voltaje de CC en las líneas de alimentación de energía.

Los chips grandes, como los PLD, con frecuencia requieren más de una conexión V_{DD} y *Ground*. En este caso es recomendable utilizar un capacitor de desacoplamiento para cada par de pines V_{DD} y *Ground* en el chip. Por ejemplo, con los PLD los fabricantes recomiendan utilizar un capacitor de 0.2 μF por cada par de pines, colocado tan cerca como sea posible del chip PLD.

Reflexiones y terminaciones

Las rutas de alambrado en una PCB actúan como cables simples en circuitos cuando la frecuencia del reloj es baja. Sin embargo, en frecuencias de reloj más altas se vuelve necesario preocu-

parse por los llamados *efectos de la línea de transmisión.* Cuando una señal se propaga a lo largo de un cable largo, se atenúa debido a la pequeña resistencia de éste, selecciona una interferencia que se manifiesta como un ruido y puede reflejarse cuando llega al final del cable. La reflexión causa un problema si su efecto no termina antes del siguiente flanco activo del reloj. El análisis de los efectos de la línea de transmisión está más allá del ámbito de este libro. Sólo mencionaremos que la reflexión de las señales puede impedirse colocando un componente de "terminación" adecuado en la línea. Esta terminación puede ser tan simple como una resistencia cuya resistencia coincide con la resistencia aparente de la línea, conocida como la *impedancia característica* de la línea. Otras formas de terminación también son posibles. Para obtener detalles de ellos se remite al lector a otros libros [17-18].

11.8.1 PRUEBAS DE LA PCB

La PCB fabricada debe probarse rigurosamente. Las fallas en el proceso de manufactura pueden provocar que algunas conexiones se rompan y otras se reduzcan por medio de una mancha de soldadura que toca dos cables que están juntos. Puede haber problemas ocasionados por errores de diseño que no se descubrieron durante el proceso respectivo. Finalmente, algunos chips y otros componentes de la PCB pueden estar defectuosos.

Encendido

El primer paso es encender la fuente de alimentación. En el peor de los casos esto puede causar la destrucción de algunos chips debido a una condición de cortocircuito fatal (en un caso extremo un encapsulado en realidad puede fundirse). Suponiendo que éste no es el caso, es esencial revisar si alguno de los chips se está calentando más de lo normal. El sobrecalentamiento es un síntoma de un problema serio que debe corregirse.

También es necesario revisar que las conexiones a la corriente y a tierra son correctas en cada chip y que el nivel de voltaje es el especificado.

Inicialización

El paso siguiente es inicializar todos los sistemas de circuitos de la PCB para alcanzar un punto de inicio predeterminado. Por lo general esto implica inicializar los flip-flops, lo cual se logra validando una línea de inicialización común. Es importante revisar que el estado inicial esté establecido correctamente.

Pruebas funcionales de nivel bajo

Puesto que los circuitos prácticos pueden ser sumamente complejos, es prudente probar la funcionalidad básica primero. Una prueba clave es verificar que las señales de control estén trabajando en forma correcta.

Si se aplica el enfoque de divide y vencerás, las funciones simples se prueban primero, seguidas de las más complejas.

Pruebas de funcionalidad completa

Una vez revisada la operación de los subcircuitos más pequeños, es necesario probar la funcionalidad de todo el sistema en la PCB. El número de errores suele depender de la meticulosidad de la simulación hecha durante el proceso de diseño. En general, es difícil simular por completo sistemas digitales grandes, por lo que es probable encontrar algunos errores en la PCB. Los errores típicos se deben a

- Errores de fabricación, como rutas de alambrado erróneas, componentes fundidos o un voltaje de la fuente de alimentación incorrecto.
- Especificaciones incorrectas.
- La mala interpretación de información sobre las hojas de datos que describen algunos chips por parte del diseñador.
- Información incorrecta de las hojas de datos provistas por el fabricante del chip.

Como ya lo mencionamos, las PCB contienen varias capas de cables. Cada capa puede tener varios miles de ellos. Encontrar y corregir los errores puede ser una tarea ardua que precise mucho tiempo, en especial si los errores comportan cables en las capas internas (lo opuesto sucede en las superiores o inferiores).

Sincronización

Es necesario revisar enseguida la sincronización del circuito. Una buena estrategia es comenzar con un reloj lento. Si el circuito funciona bien, entonces la frecuencia se aumenta en forma gradual hasta que se llega a la frecuencia de operación requerida.

Los problemas de sincronización surgen por retrasos de propagación a través de varias trayectorias en un circuito. Esos retrasos son ocasionados por las compuertas lógicas y los cables que las interconectan. Es esencial asegurar que todas las entradas de datos de los flip-flops en el circuito son estables antes que llegue la señal del reloj del flanco activo, según lo requiere el tiempo de preparación.

Fiabilidad

Se espera que un sistema digital funcione de manera confiable por un tiempo prolongado. Esta fiabilidad puede verse afectada por varios factores, como problemas de sincronización, ruido e interferencia.

La sincronización de las señales ha de brindar un margen de seguridad para permitir pequeños cambios en los retrasos de propagación. Si la sincronización es demasiado ajustada, entonces es probable que el circuito opere correctamente durante cierto periodo, pero con el tiempo fallará a causa de un error de sincronización. La sincronización de los chips puede cambiar con la temperatura, así que pueden ocurrir fallas si no se cumplen las restricciones térmicas. El enfriamiento por lo general se provee mediante ventiladores.

11.8.2 INSTRUMENTACIÓN

Las pruebas de los circuitos implementados en las PCB requieren algunos instrumentos especializados.

Osciloscopio

Los detalles de las señales individuales pueden examinarse con un osciloscopio. Este instrumento muestra la forma de onda del voltaje de una señal, con lo que se advierten los problemas potenciales respecto al retraso de propagación y al ruido. La forma de onda exhibida en un osciloscopio muestra los niveles de voltaje reales de la señal; no representa la vista simplificada de las formas de onda ideales que tienen bordes perfectamente cuadrados. Si el usuario quiere ver sólo los valores lógicos de una señal (0 o 1), entonces debe usar un tipo de instrumento diferente, llamado analizador lógico.

Analizador lógico

Mientras que un osciloscopio permite examinar al mismo tiempo algunas cuantas señales, un analizador lógico posibilita el examen de decenas o incluso cientos de señales a la vez. Toma las entradas desde un conjunto de puntos en el circuito, por medio de sondas conectadas a estos

puntos, digitaliza y despliega en una pantalla las señales detectadas en forma de ondas. Una característica poderosa del analizador lógico es que tiene la capacidad de grabar internamente una secuencia de cambios en las señales durante un periodo sustancial. Luego cualquier segmento de esta información puede mostrarse según lo desea el operador. En general, es posible registrar eventos que duran unos cuantos milisegundos, lo que implica muchos ciclos de un reloj digital normal.

Observar las formas de ondas tomadas cuando el circuito a prueba está funcionando correctamente no es útil en el proceso de depuración. Es esencial ver las formas de onda cuando un mal funcionamiento ocurre. El analizador lógico puede "desencadenarse" para registrar una serie de eventos que ocurrieron antes y después del evento "desencadenador". El usuario debe especificar cuál será éste. Supóngase que se sospecha un mal funcionamiento causado por dos señales de control, *A* y *B,* que se validan al mismo tiempo, aun cuando la especificación del diseño requiere que sean mutuamente exclusivas. Un punto de desencadenamiento útil puede establecerse cuando el AND de *A* y *B* tiene el valor de 1. Establecer eventos "desencadenadores" apropiados puede ser difícil, y para ello el usuario ha de confiar en la intuición y la experiencia.

Para usar bien un analizador lógico debe ser posible conectar las sondas a algunos puntos útiles (para propósitos de prueba) en el circuito. Por ello, es importante proporcionar estos puntos de "prueba" cuando se diseñe una PCB.

11.9 COMENTARIOS FINALES

Los productos fabricados deben probarse para asegurar que se desempeñan según lo esperado. Todas las técnicas estudiadas en este capítulo son relevantes para este tipo de pruebas. El desarrollo de pruebas y las respuestas requeridas se basa en la suposición de que los circuitos están bien diseñados. Por consiguiente, es la validez de la implementación física lo que se está probando.

Otro aspecto de las pruebas ocurre durante el proceso de diseño. El diseñador debe determinar que el circuito diseñado cumple las especificaciones. Desde el punto de vista de las pruebas, esto plantea un problema, ya que no existe ningún circuito cuyo buen estado sea demostrable que sirva para generar las pruebas deseadas. Las herramientas CAD son útiles en la derivación de pruebas para un circuito diseñado, pero no pueden determinar si éste es lo que el diseñador pretende lograr en términos de funcionalidad. Un error de diseño en general resulta en un circuito con una funcionalidad un tanto distinta a la requerida por la especificación.

Los circuitos pequeños pueden probarse completamente para revisar su funcionalidad. Un circuito combinacional puede probarse para ver si su desempeño está de acuerdo con su tabla de verdad. Un circuito secuencial puede probarse revisando las transiciones especificadas en la tabla de estado. Esto es mucho más fácil si el circuito se diseña de modo tal que sea posible aplicarle pruebas, como explicamos en la sección 11.6.1. Los circuitos grandes no pueden probarse en forma exhaustiva, pues tendría que aplicarse una gran cantidad de pruebas. En ese caso se necesita la experiencia del diseñador para determinar un conjunto manejable de pruebas que se espera demuestren la corrección del circuito.

PROBLEMAS

***11.1** Derive una tabla parecida a la de la figura 11.1*b* para el circuito de la figura P11.1 a fin de mostrar la cobertura de las diversas fallas de atascamiento en 0 y atascamiento en 1 por medio de las ocho pruebas posibles. Encuentre un conjunto de pruebas mínimo para este circuito.

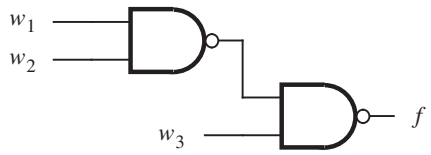


Figura P11.1 Circuito para el problema 11.1.

11.2 Repita el problema 11.1 para el circuito de la figura P11.2.

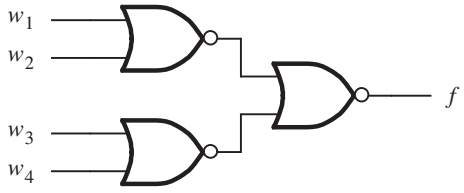


Figura P11.2 Circuito para el problema 11.2.

***11.3** Idee una prueba para distinguir entre dos circuitos que implementan las expresiones siguientes

$$f = x_1x_2x_3 + x_2\overline{x}_3x_4 + \overline{x}_1\overline{x}_2x_4 + \overline{x}_1x_3\overline{x}_4$$
$$g = (\overline{x}_1 + x_2)(x_3 + x_4)$$

11.4 Considere el circuito de la figura P11.3. Sensibilice cada trayectoria de este circuito para obtener un conjunto de pruebas completo que comprenda un número mínimo de pruebas.

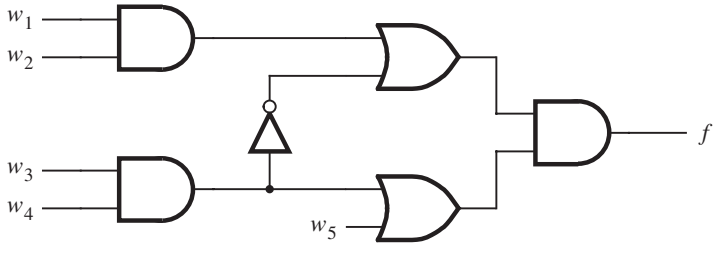


Figura P11.3 Circuito para el problema 11.4.

***11.5** Para el circuito de la figura 11.4*a,* muestre las pruebas que pueden detectar cada una de las fallas siguientes: $w_1/0$, $w_4/1$, $g/0$ y $c/1$.

11.6 Suponga que las pruebas $w_1w_2w_3w_4 = 0100$, 1010, 0011, 1111 y 0110 se eligen al azar para probar el circuito de la figura 11.3. ¿Qué porcentaje de fallas individuales se detecta usando estas pruebas?

11.7 Repita el problema 11.6 para el circuito de la figura 11.4*a*.

11.8 Repita el problema 11.6 para el circuito de la figura 11.5.

***11.9** Considere el circuito de la figura P11.4. ¿Todas las fallas individuales de atascamiento en 0 y de atascamiento en 1 de este circuito son detectables? Si no es así, explique por qué.

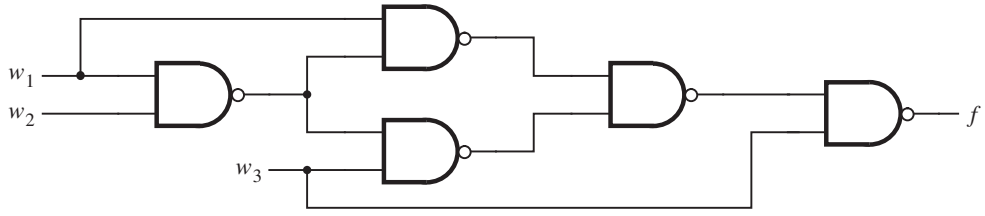


Figura P11.4 Circuito para el problema 11.9.

11.10 Demuestre que en un circuito en el que todas las compuertas tienen una carga de salida (*fan-out*) de 1, cualquier conjunto de pruebas que detecte todas las fallas individuales en los cables de entrada detecta todas las fallas individuales en el circuito completo.

***11.11** El circuito de la figura P11.5 determina la paridad de una unidad de datos de cuatro bits. Derive un conjunto de pruebas mínimo que detecte todas las fallas individuales de atascamiento en 0 y atascamiento en 1 en este circuito. ¿Su conjunto de pruebas funcionaría si las compuertas XOR se implementan utilizando el circuito de la figura 4.26*c*? ¿Su resultado puede ampliarse a un caso general que suponga unidades de datos de *n* bits?

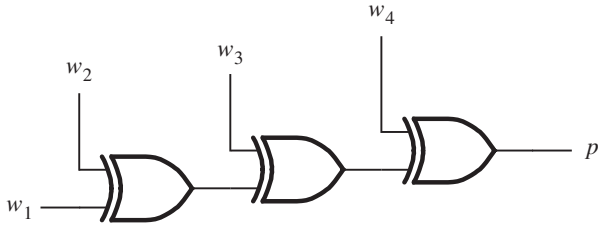


Figura P11.5 Circuito para el problema 11.11.

***11.12** Derive un conjunto de pruebas que pueda detectar todas las fallas individuales del circuito decodificador de la figura 6.16*c*.

11.13 Enumere todas las fallas individuales en el circuito de la figura 11.4*a* que pueden detectarse usando cada una de las pruebas $w_1w_2w_3w_4 = 1100$, 0010 y 0110.

11.14 Sensibilice cada trayectoria de la parte combinacional del circuito de la figura 11.12 para obtener un conjunto de pruebas completo que se componga de la menor cantidad de pruebas posible. Muestre cómo su conjunto de pruebas puede aplicarse para probar este circuito. ¿Cuántos ciclos de reloj se necesitan para realizar las pruebas necesarias?

11.15 Derive una carta ASM que represente el flujo de control necesario para probar el circuito de la figura 11.12.

11.16 El circuito de la figura 11.12 proporciona una implementación de la FSM de la figura 8.81 que puede probarse fácilmente. En el ejemplo 11.3 mostramos cómo este circuito puede examinarse al probar la parte combinacional utilizando pruebas elegidas al azar. Un método diferente para aplicar las pruebas consiste en determinar si el circuito en realidad produce la funcionalidad especificada en la tabla de estado de la figura 8.81*b*. Esto puede lograrse haciendo que el circuito pase por todas las transiciones dadas en la tabla de estado. Por ejemplo, después de aplicar la señal *Resetn* = 0, el circuito comienza en el estado *A.* Debe revisarse que el circuito se vea forzado a entrar en el estado *A* al explorar la combinación esperada $y_2 y_1 = 00$. A continuación cada transición debe revisarse. Para examinar la transición $A \rightarrow A$ si $w = 0$ es necesario hacer que la entrada *w* sea igual a 0 y permitir que la operación normal ocurra durante un ciclo de reloj al hacer $\overline{Normal}/Scan = 0$. Debe observarse el valor de la salida *z*. Esto va seguido por la exploración de los valores de y_2 y y_1 para ver si $y_1 y_2 = 00$. Al mismo tiempo, la combinación para la prueba siguiente debe explorarse. Si esta prueba implica verificar que $B \rightarrow A$ si $w = 0$, entonces la combinación de $y_2 y_1 = 01$ se introduce. Este procedimiento continúa hasta que se han revisado todas las combinaciones.
Indique en forma de tabla los valores de las señales $\overline{Normal}/Scan$, *Scan-in, Scan-out, w* y *z,* así como la transición probada, por cada ciclo de reloj necesario para realizar las pruebas completas para este circuito.

11.17 Escriba el código de VHDL que representa el circuito de la figura 11.12.

11.18 Derive una carta ASM que describa el control necesario para probar un sistema digital que utiliza la estructura BILBO de las figuras 11.18 y 11.19.

BIBLIOGRAFÍA

1. A. Miczo, *Digital Logic Testing and Simulation* (Wiley: Nueva York, 1986).
2. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. F. H. Hill y G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4a. ed. (Wiley: Nueva York, 1993).
4. Y. M. El Ziq, "Automatic Test Generation for Stuck-Open Faults in CMOS VLSI", Proc. 18th Design Automation Conf., 1981, pp. 347-354.
5. D. Baschiera y B. Courtois, "Testing CMOS: A Challenge", *VLSI Design*, octubre de 1984, pp. 58-62.
6. P. S. Moritz y L. M. Thorsen, "CMOS Circuit Testability", *IEEE Journal of Solid State Circuits* SC-21 (abril de 1986), pp. 306-309.

7. J. P. Roth, *et al*., "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits", *IEEE Transactions on Computers* EC-16, núm. 5, (octubre de 1967), pp. 567-580.
8. J. Abraham y V. K. Agarwal, "Test Generation for Digital Systems", en D. K. Pradhan, *Fault-Tolerant Computing*, vol. 1 (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. T. W. Williams y K. P. Parker, "Design for Testability—a Survey", *IEEE Transactions on Computers* C-31 (enero de 1982), pp. 2-15.
10. V. P. Nelson, H. T. Nagle, B. D. Carroll y J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
11. W. W. Peterson y E. J. Weldon Jr., *Error-Correcting Codes*, 2a. ed. (MIT Press: Boston, MA, 1972).
12. J. E. Smith, "Measures of Effectiveness of Fault Signature Analysis", *IEEE Transactions on Computers* C-29, núm. 7 (junio de 1980), pp. 510-514.
13. R. David, "Testing by Feedback Shift Register", *IEEE Transactions on Computers* C-29, núm. 7 (julio de 1980), pp. 668-673.
14. B. Koenemann, J. Mucha y G. Zwiehoff, "Built-in Logic Block Observation Techniques", Proceedings 1977 Test Conference, IEEE Pub. 79CH1609-9C, octubre de 1979, pp. 37-41.
15. A. Y. Chan, "Easy-to-Use Signature Analyzer Accurately Troubleshoots Complex Logic Circuits", *Hewlett-Packard Journal*, mayo de 1997, pp. 9-14.
16. *Test Access Port and Boundary-Scan Architecture*, estándar IEEE 1149.1, mayo de 1990.
17. *High-Speed Board Designs,* Application Note 75, Altera Corporation, enero de 1998.
18. L. Y. Levesque, "High-Speed Interconnection Techniques", Technical Report, Texas Instruments Inc., 1994.

capítulo

12

HERRAMIENTAS DE DISEÑO ASISTIDO POR COMPUTADORA

OBJETIVOS DEL CAPÍTULO

En este capítulo aprenderá a utilizar las herramientas CAD para diseñar e implementar un circuito lógico. La exposición trata las etapas de síntesis y diseño físico en un sistema CAD típico, y entre los temas expuestos se cuentan los siguientes:

- Generación de la lista de redes
- Mapeo de tecnología
- Colocación
- Enrutamiento
- Análisis de tiempo estático

Presentamos las herramientas CAD en la sección 2.9 y las hemos estudiado brevemente en otros capítulos. En este contexto, la palabra *herramienta* significa un programa de software que permite al usuario realizar una tarea específica. En este capítulo describimos con más detalle algunas de las herramientas de un sistema CAD típico, mediante un pequeño ejemplo de diseño que se procesa y mejora a medida que pasa por las diferentes etapas del flujo CAD.

12.1 SÍNTESIS

En la figura 12.1, una reproducción de la figura 2.29, se presenta un cuadro general de un sistema CAD. Se preparó una descripción del circuito deseado, en general en forma de un lenguaje de descripción de hardware como VHDL. El código de VHDL se procesa después en la etapa de síntesis del sistema CAD. La síntesis es el proceso por el que se genera un circuito lógico a partir de la especificación del usuario. En la figura 12.2 se muestran tres fases comunes que comprende el proceso de síntesis.

12.1.1 GENERACIÓN DE LA LISTA DE REDES

En la fase de *generación de la lista de redes* (*netlist*) se revisa la sintaxis del código y se informa respecto a cualesquiera errores hallados; por ejemplo, señales indefinidas, paréntesis faltantes y palabras reservadas erróneas. Una vez corregidos todos los errores se genera una lista de redes del circuito según lo determina la semántica del código de VHDL. La lista de redes utiliza expresiones lógicas para describir el circuito e incluye componentes como sumadores, flip-flops y máquinas de estado finito.

12.1.2 OPTIMACIÓN DE COMPUERTAS

La fase siguiente es la de *optimación de compuertas,* en la que se realizan los tipos de optimaciones lógicas descritos en el capítulo 4. Estas optimaciones manipulan la lista de redes para obtener un circuito equivalente, pero mejor, de acuerdo con las metas de optimación. Como vimos en la sección 2.9.2, la medida de lo que hace que un circuito sea mejor que otro puede basarse en el costo del circuito, en su velocidad de operación o en una combinación de ambos criterios.

Como ejemplo de los resultados producidos por las fases de síntesis estudiadas hasta ahora, considérese el código de VHDL para la entidad *addersubtractor* de la figura 12.3, que especifica un circuito que puede sumar o restar números de n bits y acumular el resultado en un registro. Con base en este código, la herramienta de síntesis produce una lista de redes correspondiente al circuito de la figura 12.4. Los números de entrada, $A = a_0,\ldots, a_{n-1}$ y $B = b_0,\ldots, b_{n-1}$, se colocan en los registros *Areg* y *Breg* antes de utilizarlos en operaciones de suma o resta. Estos registros sincronizan la operación del circuito si A y B se proporcionan externamente como entradas asíncronas. La entrada de control *Sel* determina el modo de operación. Si $Sel = 0$, entonces A se selecciona como una entrada para el sumador; si $Sel = 1$, entonces se selecciona el registro resultante *Zreg*. La entrada de control *AddSub* determina si la operación es una suma o una resta. Los flip-flops de la figura 12.4 para los registros *A*, *B, Sel, AddSub* y *Overflow* se infieren a partir del código de la parte inferior de la figura 12.3*a.* Los multiplexores se producen desde la entidad *mux2to1* de la figura 12.3*b*, y un sumador se genera a partir de la entidad *adderk* de la figura

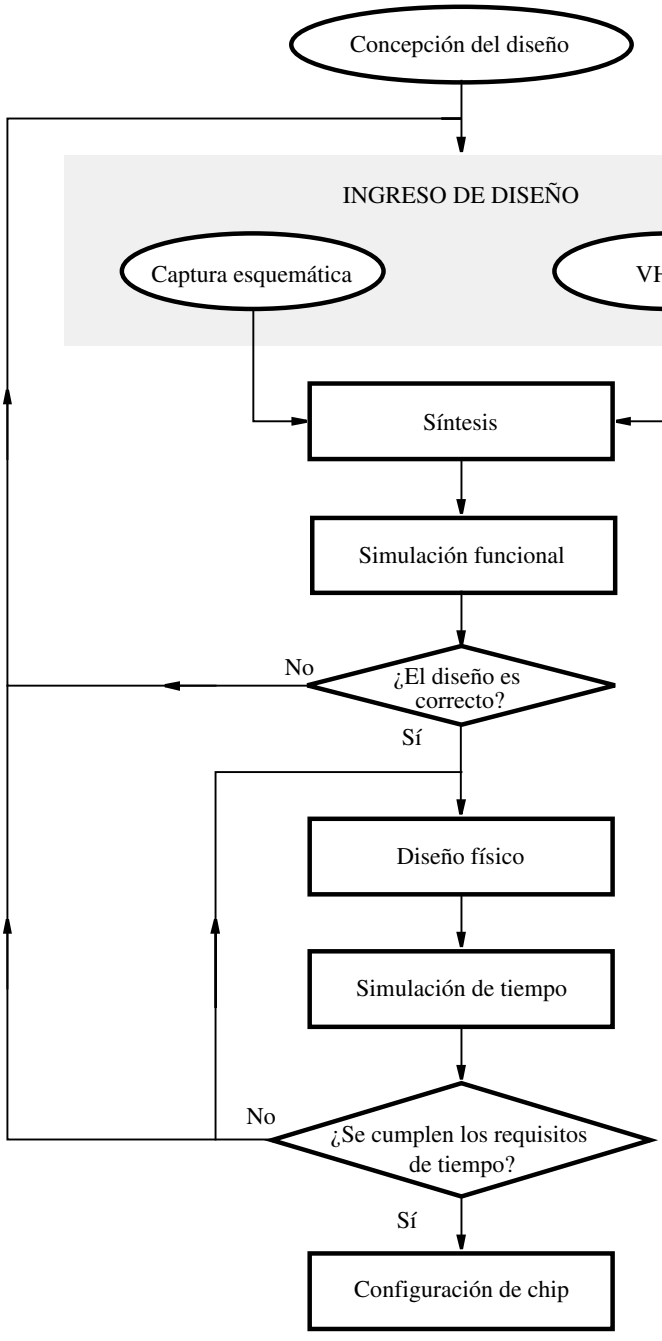


Figura 12.1 Un sistema CAD típico.

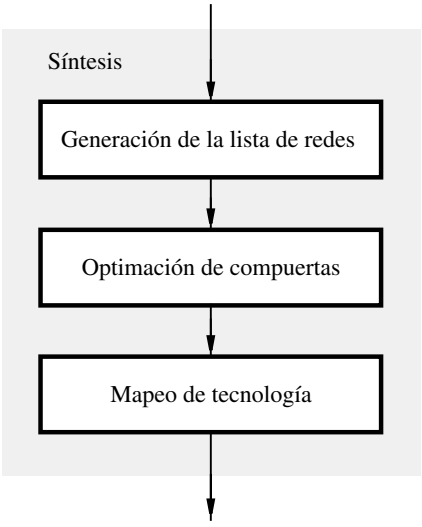


Figura 12.2 Las etapas incluidas en una herramienta de síntesis.

12.3*c*. Las compuertas OR exclusivo conectadas al registro *B* y la función XOR para la salida *Overflow* se generan a partir del código al final de la entidad *addersubtractor*.

12.1.3 MAPEO DE TECNOLOGÍA

La fase final de la síntesis es el *mapeo de tecnología.* En esta fase se determina cómo puede producirse cada uno de los componentes de la lista de redes en los recursos disponibles en el chip objetivo. Para ver los resultados del mapeo de tecnología supóngase que hemos seleccionado un FPGA para la implementación de nuestro circuito de ejemplo. En la sección 3.6.5 mostramos que un FPGA contiene un arreglo bidimensional de bloques lógicos. En la figura 3.38 se muestra el diagrama de un bloque lógico simple que contiene una tabla de consulta (LUT) de tres entradas y un flip-flop. El bloque tiene una salida, que puede seleccionarse desde la LUT o desde el flip-flop.

En la figura 12.5*a* se presenta un bloque lógico más flexible. Contiene una LUT de cuatro entradas y un flip-flop, y tiene dos salidas. Se proporciona un multiplexor para permitir la carga del flip-flop desde la LUT o directamente desde la entrada *In3.* Otro multiplexor permite que el valor almacenado en el flip-flop se retroalimente a una entrada de la LUT. Hay varias maneras o *modos* de utilizar este bloque lógico. La opción más sencilla es implementar en la LUT una función de hasta cuatro entradas y almacenar el valor de su función en el flip-flop; tanto la LUT como el flip-flop pueden proporcionar salidas desde el bloque lógico. En los incisos *b* a *e* de la figura se ilustran otros cuatro modos de usar el bloque. En los incisos *b* y *c* sólo se utiliza la LUT o el flip-flop, pero no ambos. En el inciso *d* sólo la LUT proporciona una salida para el bloque lógico y una de sus entradas se conecta al flip-flop.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY addersubtractor IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( A, B                    : IN     STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Clock, Reset, Sel, AddSub : IN     STD_LOGIC ;
           Z                       : BUFFER STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Overflow                : OUT    STD_LOGIC ) ;
END addersubtractor ;

ARCHITECTURE Behavior OF addersubtractor IS
    SIGNAL G, H, M, Areg, Breg, Zreg, AddSubR_n : STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
    SIGNAL SelR, AddSubR, carryout, over_flow : STD_LOGIC ;
    COMPONENT mux2to1
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( V, W : IN  STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
               Sel  : IN  STD_LOGIC ;
               F    : OUT STD_LOGIC_VECTOR(k−1 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT adderk
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( carryin  : IN  STD_LOGIC ;
               X, Y     : IN  STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
               S        : OUT STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
               carryout : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    PROCESS ( Reset, Clock )
    BEGIN
        IF Reset = '1' THEN
            Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
            Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
        END IF ;
    END PROCESS ;
```

. . . continúa en el inciso *b*

Figura 12.3 Código de VHDL para un circuito acumulador (inciso *a*).

En el capítulo 5 dijimos que los FPGA a menudo contienen sistemas de circuitos dedicados para la implementación de circuitos sumadores rápidos. En la figura 12.5*e* se muestra una forma de producir este sistema de circuitos. La LUT se utiliza en dos mitades; una de ellas produce la función suma de tres entradas de la LUT y la otra genera la función de acarreo de estas entradas (recuérdese que en la sección 3.6.5 vimos que una LUT de cuatro entradas se construye utilizando dos LUT de tres entradas). La función suma puede producir una salida para el bloque o almacenarse en el flip-flop, y la función de acarreo proporciona una señal de salida especial. Esta salida de acarreo se conecta directamente al bloque lógico vecino que la utiliza como entrada de

```
    nbit_adder: adderk
        GENERIC MAP ( k => n )
        PORT MAP ( AddSubR, G, H, M, carryout ) ;
    multiplexer: mux2to1
        GENERIC MAP ( k => n )
        PORT MAP ( Areg, Z, SelR, G ) ;
    AddSubR_n <= (OTHERS => AddSubR) ;
    H <= Breg XOR AddSubR_n ;
    over_flow <= carryout XOR G(n−1) XOR H(n−1) XOR M(n−1) ;
    Z <= Zreg ;
END Behavior;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
GENERIC ( k : INTEGER := 8 ) ;
    PORT ( V, W  : IN  STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
           Sel   : IN  STD_LOGIC ;
           F     : OUT STD_LOGIC_VECTOR(k−1 DOWNTO 0) ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V, W, Sel )
    BEGIN
        IF Sel = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

. . . continúa en el inciso *c*

Figura 12.3 Código de VHDL para un circuito acumulador (inciso *b*).

acarreo. Este bloque a su vez genera la etapa siguiente de la salida de acarreo y así sucesivamente. De esta manera, las conexiones directas entre los bloques lógicos vecinos se usan para formar cadenas de acarreo rápidas.

En la figura 12.6 se muestra una parte de los resultados del mapeo de tecnología para la lista de redes generada para la figura 12.4. Cada bloque lógico se resalta con un cuadro gris y tiene una etiqueta en la esquina inferior izquierda que indica cuál modo de la figura 12.5 está usándose. En la figura se observa el bit h_0 de la figura 12.4, que es producido por un bloque lógico en el modo *d.* Este bloque emplea un flip-flop para almacenar el valor de la entrada primaria b_0 e implementa una función XOR en su LUT, lo cual es necesario en las operaciones de resta para complementar el número *B.* Una entrada XOR es provista por el bloque lógico en el modo *c* que almacena en un flip-flop el valor de la entrada *AddSub*. Este flip-flop también maneja otros 15 bloques lógicos que implementan $h_1,\ldots, h_{15}$, pero estos bloques no se muestran en la figura.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adderk IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( carryin  : IN  STD_LOGIC ;
           X, Y     : IN  STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
           S        : OUT STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
           carryout : OUT STD_LOGIC ) ;
END adderk ;

ARCHITECTURE Behavior OF adderk IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + carryin ;
    S <= Sum(k−1 DOWNTO 0) ;
    carryout <= Sum(k) ;
END Behavior ;
```

Figura 12.3 Código de VHDL para un circuito acumulador (inciso *c*).

El flip-flop *AddSub* está conectado al acarreo de entrada del primer bloque lógico en el sumador. Este bloque utiliza el modo *e* para producir las salidas de la suma y del acarreo. La suma se almacena en un flip-flop que produce z_0 y el acarreo alimenta la etapa siguiente del sumador. En la figura la función de acarreo se muestra en la forma

$$c_1 = (\overline{c_0 \oplus h_0}) \cdot h_0 + (c_0 \oplus h_0) \cdot g_0$$

Esta expresión es funcionalmente equivalente a la utilizada en el capítulo 5, la cual tiene la forma $c_1 = c_0h_0 + c_0g_0 + h_0g_0$, pero representa de una manera más rigurosa cómo se construye la cadena de acarreo en un FPGA. El último bloque lógico del sumador de la figura 12.6 no utiliza su flip-flop porque la salida de la suma debe estar conectada directamente al bloque lógico que implementa la señal *Overflow*. La salida de la suma no puede proporcionarse desde la salida combinacional y de los registros al mismo tiempo, así que se necesita un bloque lógico independiente en el modo *c* para la señal z_{15}.

En la figura 12.6 se muestran sólo algunos de los bloques lógicos que una herramienta de mapeo de tecnología crearía para la implementación de nuestro circuito. En general, hay muchos métodos para realizar el mapeo, y cada uno de ellos conducirá a circuitos equivalentes pero distintos. El lector puede consultar en el material de referencia [1-3] un análisis detallado de los métodos de mapeo de tecnología.

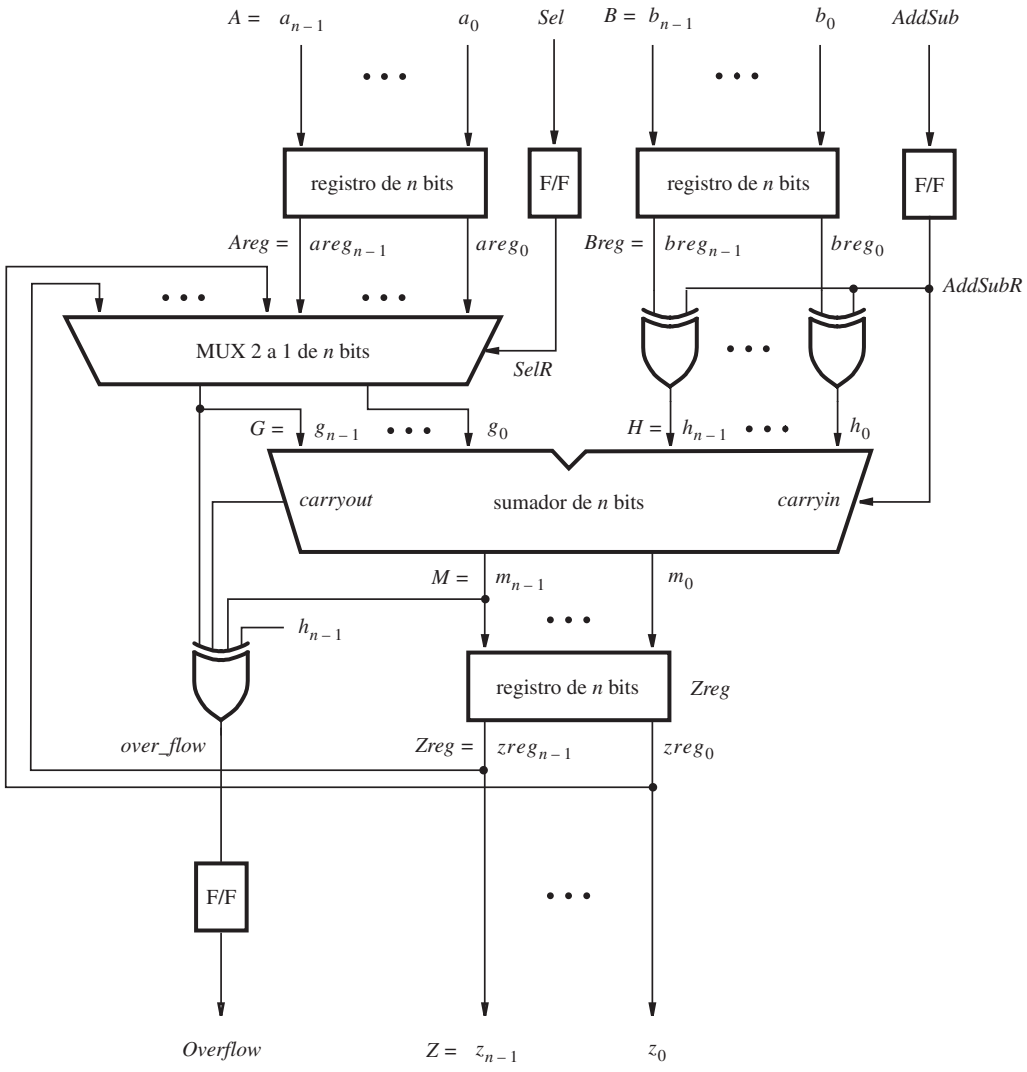
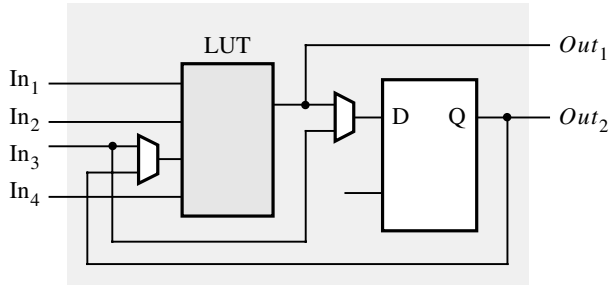


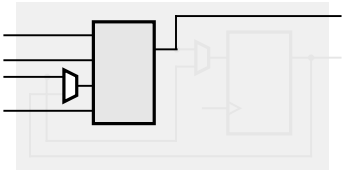
Figura 12.4 Circuito especificado por el código de la figura 12.3.

12.2 DISEÑO FÍSICO

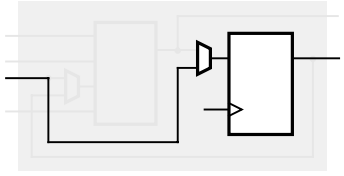
Las etapas que siguen a la síntesis en la figura 12.1 son la simulación funcional y el diseño físico. Como explicamos en la sección 2.9, la simulación funcional comprende la aplicación de patrones de prueba a la lista de redes sintetizada, así como la comprobación para ver si produce las salidas correctas. La simulación supone que no hay retrasos de propagación en el circuito,



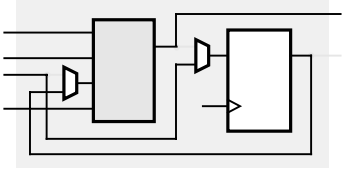
a) Un elemento lógico del FPGA



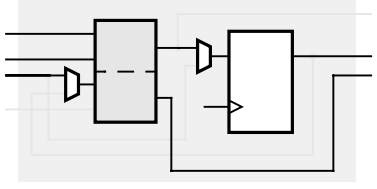
b) Modo combinacional



c) Modo síncrono



d) Modo de retroalimentación síncrona



e) Modo aritmético (síncrono)

Figura 12.5 Diferentes modos de un bloque lógico en un FPGA.

pues el objetivo consiste en evaluar la funcionalidad básica y no el tiempo. La lista de redes utilizada por un simulador funcional podría ser la versión previa al mapeo de tecnología o la versión posterior. Un ejemplo de simulación funcional que usa el software incluido en este libro se proporciona en el apéndice B, por lo que no lo estudiaremos más aquí.

Una vez que la lista de redes producida por la síntesis es funcionalmente correcta, podemos pasar a la etapa de diseño físico. En ella se determina exactamente cómo se implementará la lista de redes sintetizada en el chip objetivo. Como se ilustra en la figura 12.7, comprende tres fases: colocación, enrutamiento y análisis de tiempo estático.

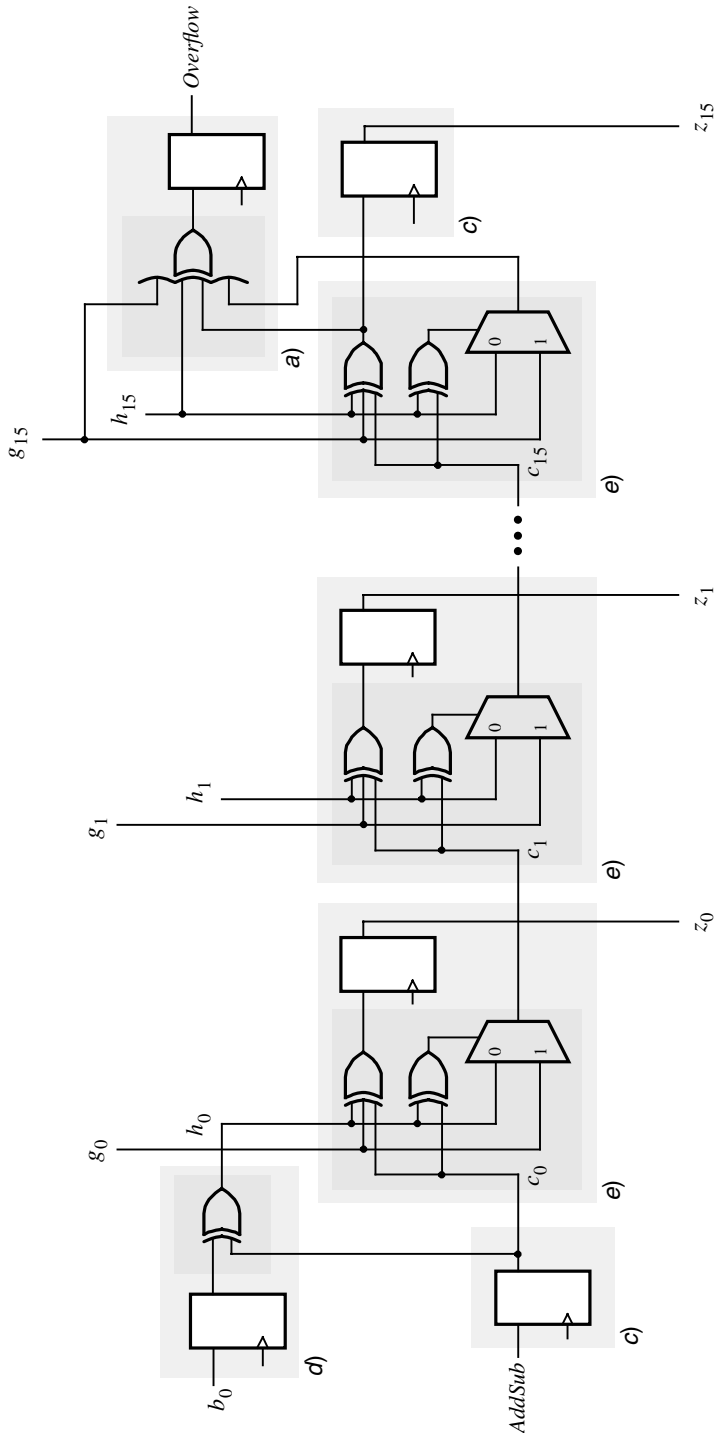


Figura 12.6 Parte del circuito de la figura 12.4 después del mapeo de tecnología.

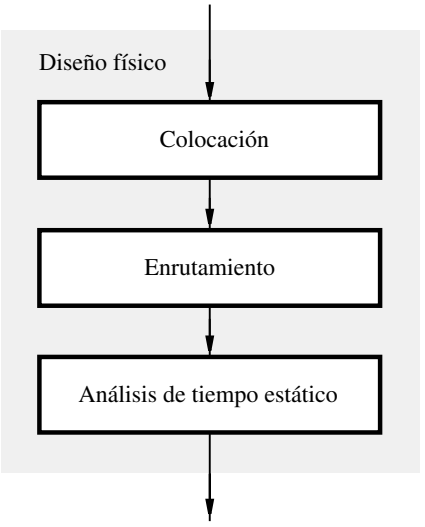


Figura 12.7 Fases del diseño físico.

12.2.1 COLOCACIÓN

En la fase de colocación se elige una ubicación en el dispositivo objetivo para cada bloque lógico de la lista de redes con mapeo de tecnología. Un ejemplo del resultado de una colocación se muestra en la figura 12.8, donde se presenta un arreglo de bloques lógicos en una pequeña parte de un chip FPGA. Los cuadros blancos representan bloques desocupados y los grises indican la colocación de bloques que implementan el circuito de la figura 12.4. Hay un total de 53 bloques lógicos en este circuito, incluidos los que aparecen en la figura 12.6. En la figura 12.8 también se muestra la colocación de algunas de las entradas primarias al circuito, las cuales se asignan a los pines a lo largo de la periferia del chip.

Para hallar una buena solución de colocación ha de considerarse una serie de ubicaciones diferentes para cada bloque lógico. Para un circuito grande, que puede contener decenas de miles de bloques, esto representa un problema arduo. A fin de apreciar la complejidad que ello entraña, considérese cuántas soluciones de colocación son posibles para un circuito. Supongamos que el circuito tiene N bloques lógicos y que debe colocarse en una FPGA que también contiene exactamente N bloques. Una herramienta de colocación tiene N opciones para la ubicación del primer bloque que selecciona. Quedan $N - 1$ opciones para el segundo bloque, $N - 2$ opciones para el tercero y así sucesivamente. La multiplicación de estas opciones arroja un total de $(N)(N - 1)\ldots (1) = N!$ soluciones de colocación posibles. Para valores aún más moderados, $N!$ es un número enorme, lo que significa que deben usarse técnicas heurísticas para hallar una buena solución mientras se considera sólo una pequeña fracción del número total de opciones. Una herramienta comercial de colocación muy común funciona construyendo una colocación inicial y moviendo después los bloques lógicos de manera iterativa. Para cada iteración se evalúa la

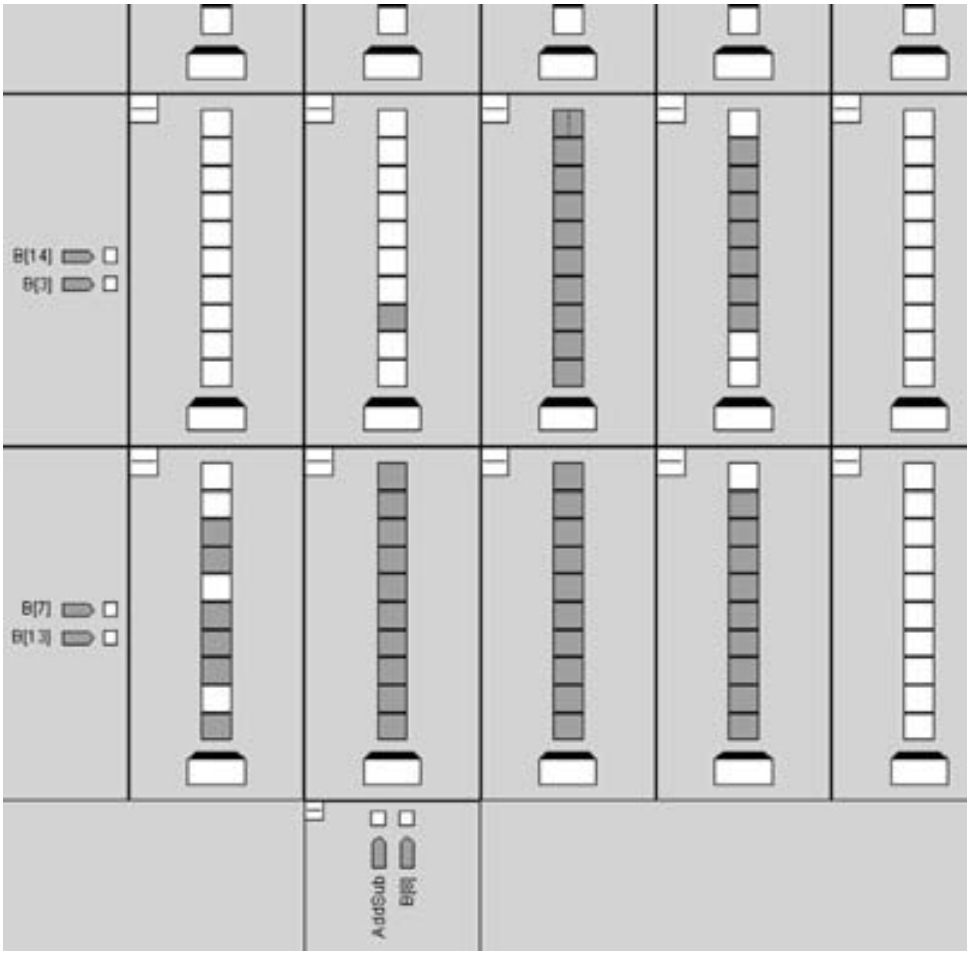


Figura 12.8 Colocación del circuito de la figura 12.6.

calidad de la solución por medio de parámetros que estiman la velocidad de operación del circuito implementado, o su costo. El problema de la colocación se ha estudiado de manera exhaustiva y se describe con detalle en las referencias [4-7].

12.2.2 ENRUTAMIENTO

Una vez que se elige la ubicación en el chip para cada bloque lógico de un circuito, en la fase de enrutamiento se conectan los bloques entre sí usando los cables que hay en el chip. Un ejemplo de una solución de enrutamiento para la colocación de la figura 12.8 se presenta en la figura 12.9. Además de mostrar los bloques lógicos, esta figura también muestra algunos de los cables del

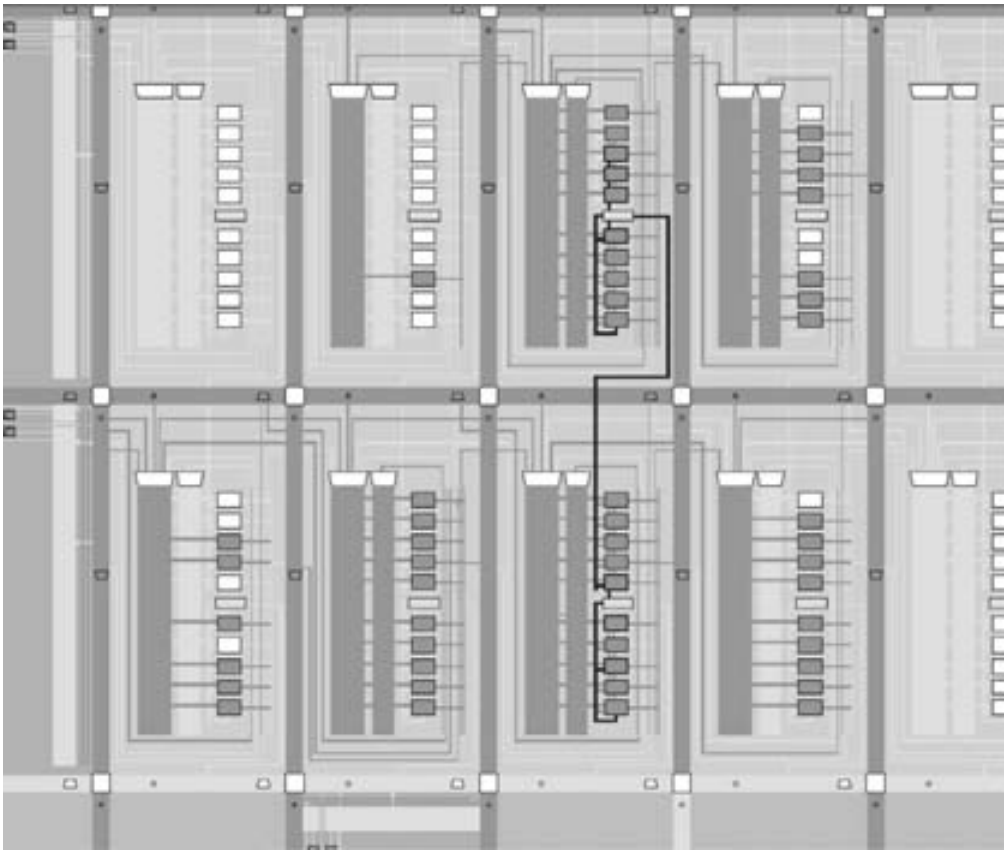


Figura 12.9 Enrutamiento para la colocación de la figura 12.8.

chip. Los cables que el circuito implementado está utilizando aparecen sombreados en gris. En la figura se describen los dos cables individuales, los cuales pueden tener varias longitudes, y los grupos de cables, que se muestran como rectángulos sombreados en gris. La herramienta CAD de enrutamiento intenta hacer el mejor uso de varios tipos de cables, por ejemplo, conexiones eficientes para cadenas de acarreo. En la figura 12.9 se muestra un ejemplo de la ruta de la cadena de acarreo de la figura 12.6. Las líneas negras resaltan los cables de esa cadena, los cuales se conectan a través de las etapas del sumador y que terminan en el registro *Overflow*. Un estudio detallado de las herramientas de enrutamiento se halla en las referencias [3], [5-6] y [8].

12.2.3 ANÁLISIS DE TIEMPO ESTÁTICO

Después que el enrutamiento se completa se conocen los retrasos de tiempo para el circuito implementado, ya que el sistema CAD calcula éstos para todos los bloques y cables del chip. Una herramienta de análisis de tiempo estático examina esta información de retraso y produce un

conjunto de tablas que cuantifican el rendimiento del circuito. Un ejemplo de un análisis de tiempo se proporciona en la tabla 12.1, que enumera cuatro parámetros: $f_{máx}$, t_{su}, t_{co} y t_h. El valor $f_{máx}$ especifica la frecuencia de operación máxima del *reloj* del circuito. Este valor está determinado por la ruta con el retraso de propagación más prolongado, a menudo llamada *trayectoria crítica,* entre cualquiera de dos flip-flops del circuito. Como expusimos en la sección 10.3, el retraso de trayectoria debe representar los retardos a través de bloques lógicos y cables, así como los parámetros de retraso de registro (t_{rd}) y de preparación (t_{su}) del flip-flop. En nuestro ejemplo, el retraso de ruta crítica es $1/261.1 \times 10^6 = 3.83$ ns. Las últimas dos columnas en la fila $f_{máx}$ muestran que la trayectoria empieza en el flip-flop *AddSub* y termina en el flip-flop *Overflow* de la figura 12.6.

La mayor parte de los sistemas CAD permite a los usuarios especificar los requisitos de tiempo para su circuito. En la tabla 12.1 supusimos que el usuario ha especificado que el *reloj* del circuito debe funcionar correctamente hasta una frecuencia de 200 MHz. La diferencia entre este requisito y el resultado obtenido por las herramientas CAD se conoce como *tolerancia* (*slack*). En la tabla, el requisito es que los retrasos de propagación no deben exceder $1/200 \times 10^6 = 5$ ns; el resultado es 3.83 ns, lo que da un valor de tolerancia de 1.17 ns. Esta tolerancia positiva signi-fica que las restricciones se cumplen con cierto espacio de sobra. Si el resultado obtenido tuvo una tolerancia negativa, entonces los requisitos del usuario no se habrán cumplido y será preciso modificar el código de VHDL o los parámetros usados en la herramienta CAD para tratar de satisfacer las restricciones.

Las otras filas en la tabla 12.1 muestran los resultados del tiempo para las entradas y salidas primarias del diseño. El resultado t_{su} indica que el requisito de preparación del peor caso es 2.356 ns desde el pin b_0 hasta el flip-flop $breg_0$. Este parámetro significa que la señal b_0 debe tener un valor estable mínimo de 2.356 ns antes de cada flanco activo de la señal de reloj en su pin asignado. Puesto que el diseñador especificó un requisito de preparación del peor caso de 10 ns, el resultado obtenido significa que el circuito implementado lo excede por un valor de tolerancia de 7.644 ns. El retraso de reloj a la salida del peor caso para nuestro circuito es 6.772 ns, desde el flip-flop $zreg_0$ hasta el pin z_0. Esto significa que el retraso de propagación desde un flanco activo de la señal de reloj en su pin correspondiente a un cambio correspondiente en la señal z_0 en su pin es 6.772 ns. Como la restricción del diseñador especifica que se permite un t_{co} de 10 ns, la tolerancia disponible es 3.228 ns.

En la última fila de la tabla 12.1 se presenta un tiempo de espera máximo de 0.24 ns, para la ruta desde el pin b_1 al flip-flop $breg_1$. Por consiguiente, la señal en el pin b_1 debe mantener un valor estable durante al menos 0.24 ns después de cada flanco activo del reloj en el pin del reloj. Suponemos que no se estableció ninguna restricción para este parámetro, por lo que no se muestra ningún valor de tolerancia.

Tabla 12.1 Resumen de los resultados del análisis de tiempo estático.

Parámetro	Real	Requerido	Tolerancia	Desde	Hasta
$f_{máx}$	261.1 MHz	200 MHz	1.17 ns	*AddSub*	*Overflow*
t_{su}	2.356 ns	10.0 ns	7.644 ns	b_0	$breg_0$
t_{co}	6.772 ns	10.0 ns	3.228 ns	$zreg_0$	z_0
t_h	0.240 ns	N/A	N/A	b_1	$breg_1$

En la tabla 12.1 se enumeran sólo las rutas del peor caso para f_{max}, t_{su}, t_{co} y t_h. El circuito implementado tendrá otra serie de rutas distintas con retrasos más pequeños y valores de tolerancia mayores. Una herramienta de análisis de tiempo estático en general proporciona tablas adicionales por cada parámetro, en las que se enumeran más rutas.

La etapa final del flujo CAD de la figura 12.1 es la simulación de tiempo. En el apéndice C mostramos cómo se realiza la simulación de tiempo aplicando patrones de prueba al circuito implementado y observando su conducta tanto funcional como en tiempo.

12.3 COMENTARIOS FINALES

En este capítulo explicamos brevemente un flujo de diseño típico hecho posible gracias a herramientas CAD poderosas. Consideramos sólo el subconjunto más importante de herramientas disponibles en sistemas CAD comerciales. Para aprender más el lector puede consultar las referencias [1-8], o visitar los sitios web de proveedores de herramientas CAD. En la tabla 12.2 se enumeran algunos de los principales vendedores de herramientas CAD, al tiempo que se consignan sus direcciones web y los nombres de algunos productos populares.

Tabla 12.2 Principales productos de herramientas CAD.

Nombre del vendedor	Dirección web	Producto
Altera	altera.com	Quartus II
Mentor Graphics	mentorgraphics.com	ModelSim, Precision
Synplicity	synplicity.com	Synplify
Synopsys	synopsys.com	Design Compiler, VCS
Xilinx	xilinx.com	ISE

BIBLIOGRAFÍA

1. R. Murgai, R. Brayton, A. Sangiovanni-Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays* (Kluwer Academic Publishers, 1995).
2. J. Cong y Y. Ding, *FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs* (en IEEE Transactions on Computer-aided Design 13 (1), enero de 1994).
3. S. Brown, R. Francis, J. Rose, Z. Vranesic, *Field-Programmable Gate Arrays* (Kluwer Academic Publishers, 1995).
4. M. Breuer, *A Class of Min-cut Placement Algorithms* (en Design Automation Conference, páginas 284-290, IEEE/ACM, 1977).

5. Carl Sechen, *VLSI Placement and Global Routing Using Simulated Annealing* (Kluwer Academic Publishers, 1988).
6. V. Betz, J. Rose y A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs* (Kluwer Academic Publishers, 1999).
7. M. Sarrafzadeh, M. Wang y X. Yang, *Modern Placement Techniques* (Kluwer Academic Publishers, 2003).
8. L. McMurchie y C. Ebeling, *PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs* (en International Symposium on Field Programmable Gate Arrays, Monterey, Ca., Feb. 1995).

apéndice

A

Referencia de VHDL

En este apéndice se describen las funciones de VHDL que se utilizan en este libro, lo que significa que sirve como material de consulta para el lector. Por consiguiente, sólo se ofrecen descripciones breves con ejemplos. Recomendamos al lector que primero estudie la introducción a VHDL en las secciones 2.10 y 4.12.

En cierta forma VHDL utiliza una sintaxis poco usual para describir los circuitos lógicos. La razón principal de ello es que VHDL se pensó originalmente como un lenguaje para la documentación y simulación de circuitos, no para describir los circuitos para la síntesis. Este apéndice no pretende ser un manual completo de VHDL. Si bien estudiamos casi todas sus funciones útiles en la síntesis de circuitos lógicos, no analizamos ninguna de las que sólo sirven para su simulación o para otros propósitos. Aun cuando las funciones omitidas no son necesarias para ninguno de los ejemplos expuestos en la obra, un lector que desee aprender más del uso de VHDL puede remitirse a libros especializados [1-7].

Cómo *no* escribir código en VHDL

En la sección 2.10 mencionamos el problema más común con que se enfrentan los diseñadores que empiezan a escribir código en VHDL: tienden a escribir código parecido al de un programa de computadora, con muchas variables y ciclos. Es difícil determinar qué circuito lógico producirán las herramientas CAD cuando sinteticen ese código. Este libro contiene más de 150 ejemplos de código completo de VHDL que representan una amplia variedad de circuitos lógicos. En todos esos ejemplos el código se relaciona fácilmente con el circuito lógico descrito. Se aconseja al lector adoptar el mismo estilo al escribir código. Una buena pauta general es suponer que si el diseñador no puede determinar de inmediato qué circuito lógico describe el código de VHDL, entonces es probable que las herramientas CAD no sinteticen el circuito que el diseñador está tratando de describir.

Como VHDL es un lenguaje complejo, los errores en la sintaxis y en el uso son muy frecuentes. Algunos problemas que nuestros estudiantes encuentran, como diseñadores novatos que son, se enumeran al final de este apéndice, en la sección A.11. El lector puede hallar útil examinarlos en un esfuerzo por evitarlos cuando escriba código.

Una vez que se escribe el código de VHDL para un diseño en particular, es útil analizar el circuito sintetizado resultante con las herramientas CAD. Hay mucho que aprender de VHDL, circuitos lógicos y síntesis lógica al estudiar los circuitos que las herramientas CAD producen automáticamente.

A.1 DOCUMENTACIÓN EN EL CÓDIGO DE VHDL

La documentación puede incluirse en el código de VHDL mediante comentarios. Dos guiones seguidos, '--', indican el principio del comentario. El compilador de VHDL ignora el texto en una línea después de esos caracteres.

Ejemplo A.1

```
-- éste es un comentario en VHDL
```

A.2 OBJETOS DE DATOS

La información se representa en el código de VHDL como objetos de datos. Se proporcionan tres tipos de objetos de datos: señales, constantes y variables. Para describir los circuitos lógicos, los objetos de datos más importantes son las señales, pues representan las señales lógicas (cables) en el circuito. Las constantes y variables a veces también son útiles para describir los circuitos, pero se usan poco.

A.2.1 NOMBRES DE OBJETOS DE DATOS

Las reglas para especificar los nombres de objetos de datos son simples: puede usarse cualquier carácter alfanumérico, así como el carácter de guión bajo '_'. Hay cuatro salvedades. Un nombre no puede ser una palabra reservada de VHDL, debe comenzar con una letra, no puede terminar con un guión bajo '_' y no puede tener dos guiones bajos '_' seguidos. De esta manera, son ejemplos de nombres permitidos *x, x*1*, x_y* y *Byte.* Algunos ejemplos de nombres no permitidos son 1*x, _y, x_ _y* y *entity.* El último nombre no está permitido porque es una palabra reservada de VHDL. Es importante notar que este lenguaje no distingue mayúsculas de minúsculas; por tanto, *x* es lo mismo que *X* y ENTITY que *entity.* Para que los ejemplos de código VHDL sean más legibles, en este libro usamos letras mayúsculas en todas las palabras reservadas.

A fin de evitar una confusión al usar la palabra *signal*, que significa ya sea un objeto de datos de VHDL o una señal lógica en un circuito, a veces escribimos los objetos de datos como SIGNAL.

A.2.2 VALORES Y NÚMEROS DEL OBJETO DE DATOS

Usamos objetos de datos SIGNAL para representar señales lógicas individuales en un circuito, señales múltiples y números binarios (enteros). El valor de un objeto SIGNAL individual se especifica utilizando apóstrofos, como en '0' o '1'. El valor de un objeto SIGNAL multibit se proporciona con comillas. Un ejemplo de un valor SIGNAL de cuatro bits es "1001", y de un valor de ocho bits es "10011000". Las comillas también se emplean para indicar un número binario. Por consiguiente, "1001" representa los cuatro valores SIGNAL '1', '0', '0', '1', pero también puede significar el entero $(1001)_2 = (9)_{10}$. Los enteros pueden especificarse en decimales si no se utilizan comillas, como en 9 o 152. Los valores de objetos de datos CONSTANT o VARIABLE se especifican de la misma forma que los objetos de datos SIGNAL.

A.2.3 OBJETOS DE DATOS SIGNAL

Los objetos de datos SIGNAL representan las señales lógicas, o cables, en un circuito. Hay tres lugares donde es posible declarar las señales en el código de VHDL: en una declaración de entidad (véase la sección A.4.1), en la sección declarativa de una arquitectura (véase la sección A.4.2) y en la sección declarativa de un paquete (véase la sección A.5). Una señal debe declararse con un *tipo* asociado como sigue:

```
SIGNAL signal_name : type_name ;
```

La variable *type_nombre* de la señal determina los valores legales que la señal puede asumir y sus usos lícitos en el código de VHDL. En esta sección describimos 10 tipos de señales: BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, SIGNED, UNSIGNED, INTEGER, ENUMERATION y BOOLEAN.

A.2.4 TIPOS BIT Y BIT_VECTOR

Estos tipos están predefinidos en los estándares de VHDL del IEEE 1076 y 1164. Por tanto, no se requiere una biblioteca para utilizarlos en el código. Los objetos del tipo BIT pueden tener los valores '0' o '1'. Un objeto de tipo BIT_VECTOR es un arreglo lineal de objetos BIT.

Ejemplo A.2

```
SIGNAL x1   : BIT ;
SIGNAL C    : BIT_VECTOR (1 TO 4) ;
SIGNAL Byte : BIT_VECTOR (7 DOWNTO 0) ;
```

Las señales *C* y *Byte* ilustran dos maneras posibles de definir un objeto de datos de múltiples bits. La sintaxis "lowest_index TO highest_index" es útil para una señal multibit que es un simple arreglo de bits. En la señal *C* el bit más significativo (el del extremo izquierdo) se referencia usando lowest_index, y el bit menos significativo (el del extremo derecho) se referencia con highest_index. La sintaxis "highest_index DOWNTO lowest_index" es útil si la señal representa un número binario. En este caso el bit más significativo (el del extremo izquierdo) tiene el índice highest_index y el bit menos significativo (el del extremo derecho) tiene el índice lowest_index.

La señal multibit *C* representa cuatro objetos BIT. Puede utilizarse como una sola cantidad de cuatro bits o puede hacerse referencia a cada bit de manera individual. La sintaxis para esto último es *C*(1), *C*(2), *C*(3) o *C*(4). Una instrucción de asignación como

```
C <= "1010" ;
```

da como resultado *C*(l) = 1, *C*(2) = 0, *C*(3) = 1 y *C*(4) = 0.

La señal *Byte* comprende ocho objetos BIT. La instrucción de asignación

```
Byte <= "10011000" ;
```

da como resultado *Byte*(7) = 1, *Byte*(6) = 0 y así sucesivamente hasta *Byte*(0) = 0.

A.2.5 TIPOS STD_LOGIC Y STD_LOGIC_VECTOR

El tipo STD_LOGIC se añadió al estándar de VHDL en el IEEE 1164. Ofrece mayor flexibilidad que el tipo BIT. Para utilizarlo debemos incluir las instrucciones

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

que proporcionan acceso al paquete *std_logic_1164*, el cual define el tipo STD_LOGIC. En la sección A.5 describimos los paquetes de VHDL. En general, se utilizan como un lugar para almacenar código de VHDL; por ejemplo, el código que define un tipo, el cual puede utilizarse después en otros archivos de código fuente. Los valores siguientes son legales para el objeto de datos STD_LOGIC: 0, 1, Z, −, L, H, U, X y W. Sólo los primeros cuatro son útiles para la síntesis de los circuitos lógicos. El valor Z representa una impedancia alta; y − significa una condición "no-importa". El valor L representa una "señal débil 0", H una "señal débil 1", U indica "sin inicializar", X significa "desconocido" y W quiere decir "señal débil desconocida". El tipo STD_LOGIC_VECTOR representa un arreglo de objetos STD_LOGIC.

Ejemplo A.3

```
SIGNAL x1, x2, Cin, Cout, Sel  : STD_LOGIC ;
SIGNAL C                       : STD_LOGIC_VECTOR (1 TO 4) ;
SIGNAL X, Y, S                 : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

Los objetos STD_LOGIC se emplean con frecuencia en las expresiones lógicas del código de VHDL. Las señales STD_LOGIC_VECTOR pueden usarse como números binarios en los circuitos aritméticos si se incluye en el código la instrucción

```
USE ieee.std_logic_signed.all ;
```

El paquete *std_logic_signed* especifica que es legal utilizar las señales STD_LOGIC_VECTOR con operadores aritméticos, como + (véase la sección A.7.1). El compilador de VHDL debe generar un circuito que funcione para números con signo. Una alternativa consiste en utilizar el paquete *std_logic_unsigned.* En este caso el compilador debe generar un circuito que funcione para números sin signo.

A.2.6 TIPOS STD_ULOGIC

En este libro usamos el tipo STD_LOGIC en la mayor parte de los ejemplos de código de VHDL. Este tipo en realidad es un *subtipo* del tipo STD_ULOGIC. Las señales que tienen el tipo STD_ULOGIC pueden tomar los mismos valores que las señales STD_LOGIC que se han

estado usando. La única diferencia entre STD_ULOGIC y STD_LOGIC se relaciona con el concepto de *función de resolución.* En VHDL una función de resolución se utiliza para determinar qué valor debe tomar una señal si hay dos fuentes para la misma. Por ejemplo, dos buffers triestado podrían tener sus salidas conectadas a una señal *x.* En algún momento, uno de ellos podría producir el valor de salida 'Z' y el otro el valor 1. Para determinar que el valor de *x* debe ser 1 en este caso se emplea una función de resolución. El tipo STD_LOGIC permite varias fuentes para una señal; resuelve el valor correcto utilizando una función de resolución que se proporciona como parte del paquete *std_logic_1164*. El tipo STD_ULOGIC no permite que las señales tengan varias fuentes. Hemos presentado STD_ULOGIC para ser congruentes, pero no se emplea en el libro.

A.2.7 TIPOS SIGNED Y UNSIGNED

Los paquetes *std_logic_signed* y *std_logic_unsigned* mencionados en la sección A.2.5 recurren a otro paquete, llamado *std_logic_arith,* que define el tipo de circuito que ha de usarse para implementar operadores aritméticos como +. El paquete *std_logic_arith* define dos tipos de señales, SIGNED y UNSIGNED. Estos tipos son idénticos al tipo STD_LOGIC_VECTOR, ya que representan un arreglo de señales STD_LOGIC. El propósito de los tipos SIGNED y UNSIGNED es permitir que el usuario indique en el código de VHDL qué tipo de representación numérica se está usando. El tipo SIGNED se utiliza en código para circuitos que manejan números con signo (complemento a 2) y el tipo UNSIGNED en código que emplea números sin signo.

Ejemplo A.4

Suponga que A y B son señales con el tipo SIGNED. Ahora suponga que se asigna a A el valor "1000" y a B "0001". VHDL proporciona operadores relacionales (véase la tabla A.1 en la sección A.3) que pueden ocuparse para comparar los valores de dos señales. La comparación $A < B$ se evalúa como verdadera porque los valores con signo son $A = -8$ y $B = 1$. Por otro lado, si A y B están definidos con el tipo UNSIGNED, entonces $A < B$ se evalúa como falsa porque los valores sin signo son $A = 8$ y $B = 1$.

El paquete *std_logic_signed* especifica que las señales STD_LOGIC_VECTOR deben tratarse como señales SIGNED. De forma similar, el paquete *std_logic_unsigned* indica que las señales STD_LOGIC_VECTOR han de tratarse como señales UNSIGNED. Se tiene una opción más bien arbitraria cuando hay que decidir si el código se escribe usando señales STD_LOGIC_VECTOR junto con los paquetes *std_logic_signed* o *std_logic_unsigned*, o utilizando señales SIGNED y UNSIGNED con el paquete *std_logic_arith*.

El paquete *std_logic_arith*, y por consiguiente los paquetes *std_logic_signed* y *std_logic_unsigned*, en realidad no forman parte de los estándares de VHDL. Synopsys Inc., un fabricante de software CAD, proporciona estos estándares. Sin embargo, esos paquetes se incluyen en el grueso de los sistemas CAD que soportan VHDL y son muy empleados en la práctica.

A.2.8 TIPO INTEGER

El estándar de VHDL define el tipo INTEGER para usarlo con operadores aritméticos. En esta obra en lo general se prefiere el tipo STD_LOGIC_VECTOR en el código para circuitos aritméticos, aunque el tipo INTEGER se utiliza de manera ocasional. Una señal INTEGER representa un número binario. El código no proporciona de manera expresa el número de bits de la señal, como lo hace para las señales STD_LOGIC_VECTOR. De forma predeterminada, una señal INTEGER tiene 32 bits y puede representar números desde $-(2^{31} - 1)$ hasta $2^{31} - 1$. Éste es un número menor que los límites normales del complemento a 2; la razón es simplemente que el estándar de VHDL especifica un número igual de números negativos y positivos. Los enteros con menos bits también pueden declararse por medio de la palabra reservada RANGE.

Ejemplo A.5

```
SIGNAL X : INTEGER RANGE −127 TO 127 ;
```

Esto define a *X* como un número con signo de ocho bits.

A.2.9 TIPO BOOLEAN

Un objeto de tipo BOOLEAN puede tener los valores TRUE o FALSE; TRUE equivale a 1 y FALSE a 0.

Ejemplo A.6

```
SIGNAL Flag : Boolean ;
```

A.2.10 TIPO ENUMERATION

Una señal de tipo ENUMERATION es aquélla para la que el usuario especifica los valores posibles. La forma general de un tipo ENUMERATION es

```
TYPE enumerated_type_name IS (name {, name}) ;
```

Las llaves indican que pueden incluirse uno o más elementos adicionales. Las utilizamos de esta manera en varias partes del apéndice. El uso más común del tipo ENUMERATION es para indicar los estados de una máquina de estado finito.

Ejemplo A.7

```
TYPE State_type IS (stateA, stateB, stateC) ;
SIGNAL y : State_type ;
```

Esto declara una señal llamada *y,* para la cual los valores legales son *stateA, stateB* y *stateC.* Cuando el compilador de VHDL traduce el código, automáticamente asigna patrones de bits (códigos) para representar *stateA, stateB* y *stateC.*

A.2.11 OBJETOS DE DATOS CONSTANT

Un objeto de datos CONSTANT es aquel cuyo valor no puede cambiarse. A diferencia de un objeto SIGNAL, un objeto CONSTANT no representa un cable en un circuito. La forma general de una declaración CONSTANT es

```
CONSTANT constant_name : type_name := constant_value ;
```

El propósito de una constante es mejorar la legibilidad del código mediante el nombre de la constante en lugar de un valor o número.

Ejemplo A.8

```
CONSTANT Zero : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000" ;
```

Por tanto la palabra *Zero* puede usarse en el código para indicar el valor constante "0000".

A.2.12 OBJETOS DE DATOS VARIABLE

Un objeto de datos VARIABLE, a diferencia de SIGNAL, no necesariamente representa un cable en un circuito. Los objetos de datos VARIABLE a veces sirven para almacenar los resultados de los cálculos y para las variables de índice en los ciclos. Daremos algunos ejemplos en la sección A.9.7.

A.2.13 CONVERSIÓN DE TIPOS

VHDL es un lenguaje muy riguroso en lo que al manejo de tipos se refiere, lo cual significa que no permite que el valor de una señal con un tipo se asigne a otra señal con un tipo distinto. Incluso para las señales que parecen intuitivamente compatibles, como BIT y STD_LOGIC, no se permite el uso de los dos tipos juntos. Para evitar este problema, en la obra en general usamos

sólo los tipos STD_LOGIC y STD_LOGIC_VECTOR. Cuando es necesario utilizar código que tenga una mezcla de tipos, pueden usarse las funciones de conversión de tipos para traducir de un tipo a otro.

Supóngase que *X* se definió como una señal STD_LOGIC_VECTOR de ocho bits y *Y* es una señal INTEGER definida con el límite de 0 a 255. Un ejemplo de una función de conversión que permite que el valor de *Y* se asigne a *X* es

```
X <= CONV_STD_LOGIC_VECTOR(Y, 8) ;
```

Esta función de conversión tiene dos parámetros: el nombre de la señal que se va a convertir y el número de bits en *X*. La función se proporciona como parte del paquete *std_logic_arith*; por consiguiente debe incluirse en el código utilizando las cláusulas LIBRARY y USE apropiadas.

A.2.14 ARREGLOS

Mencionamos antes que los tipos BIT_VECTOR y STD_LOGIC_VECTOR son arreglos de señales BIT y STD_LOGIC respectivamente. Las definiciones de estos arreglos, que se proveen como parte de los estándares de VHDL, son

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE < >) OF BIT ;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE < >) OF STD_LOGIC ;
```

El tamaño de los arreglos no se establece en las definiciones; la sintaxis (NATURAL RANGE < >) tiene el efecto de permitir que el usuario establezca el tamaño del arreglo cuando declare un objeto de datos de cualquier tipo. Los arreglos de cualquier tipo pueden ser definidos por el usuario. Por ejemplo

```
TYPE Byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC ;
SIGNAL X : Byte ;
```

declara la señal *X* con el tipo *Byte,* que es un arreglo de ocho elementos de objetos de datos STD_LOGIC.

Un ejemplo que define un arreglo bidimensional es

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL R : RegArray ;
```

Este código define a *R* como un arreglo con cuatro elementos. Cada elemento es una señal STD_LOGIC_VECTOR de ocho bits. La sintaxis $R(i)$, donde $3 \geq i \geq 0$ se usa para referirse al elemento *i* de un arreglo. La sintaxis $R(i)(j)$, donde $7 \geq j \geq 0$, se utiliza para referirse a un bit en el arreglo $R(i)$. Este bit tiene el tipo STD_LOGIC. Un ejemplo que emplea el tipo *RegArray* se muestra en la sección 10.2.6.

A.3 OPERADORES

VHDL proporciona una serie de operadores útiles para la síntesis, la simulación y la documentación de los circuitos lógicos. En la sección 6.6.8 estudiamos los operadores que se utilizan con fines de síntesis. Los enumeramos según su funcionalidad. El estándar de VHDL agrupa en clases formales todos los operadores, como se muestra en la tabla A.1. Los operadores de una clase tienen la misma precedencia. La precedencia de clases se indica en la tabla. Obsérvese que el operador NOT está en la clase Miscellaneous en vez de en la clase Logical. Por tanto, NOT tiene mayor precedencia que AND y OR.

En una expresión lógica, los operadores de una misma clase se evalúan de izquierda a derecha. Los paréntesis siempre deben usarse para asegurar la interpretación correcta de la expresión. Por ejemplo, la expresión

```
x1 AND x2 OR x3 AND x4
```

no posee el significado $x_1x_2 + x_3x_4$ que se esperaría porque AND no tiene precedencia sobre OR. Para obtener el significado buscado debe escribirse así

```
(x1 AND x2) OR (x3 AND x4)
```

Tabla A.1 Operadores de VHDL.

	Clase de operador	Operador
Precedencia mayor	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, −
	Adding	+, −, &
	Relational	+, /=, <, <=, >, >=
Precedencia menor	Logical	AND, OR, NAND, NOR, XOR, XNOR

A.4 ENTIDAD DE DISEÑO DE VHDL

Un circuito o subcircuito descrito con código de VHDL se llama *entidad de diseño* o, simplemente, *entidad*. En la figura A.1 se muestra la estructura general de una entidad. Ésta tiene dos partes principales: la *declaración de entidad* (ENTITY), que especifica las señales de entrada y de salida para la entidad, y la arquitectura, que proporciona los detalles del circuito.

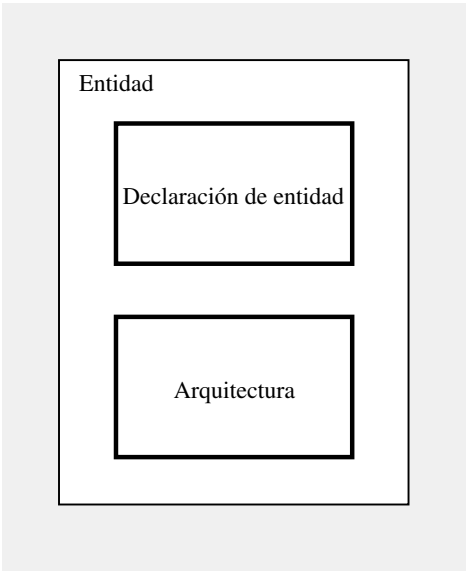


Figura A.1 Estructura general de una entidad de diseño de VHDL.

A.4.1 DECLARACIÓN ENTITY

Las señales de entrada y salida de una entidad se especifican usando la declaración ENTITY, como se indica en la figura A.2. El nombre de la entidad puede ser cualquiera que sea legal en VHDL. Los corchetes indican un elemento opcional. Las señales de entrada y salida se especifican por medio de la palabra reservada PORT. Cada puerto, sin importar si es una señal de entrada, salida o bidireccional, se indica por su *modo*. Los modos disponibles se resumen en la tabla A.2. Si no se establece el modo de un puerto, se presupone que tiene el modo IN.

A.4.2 ARQUITECTURA

Una arquitectura (ARCHITECTURE) provee los detalles del circuito para una entidad. La estructura general de una arquitectura se muestra en la figura A.3. Consta de dos partes principales: la *región declarativa* y el *cuerpo de arquitectura.* La región declarativa precede a la palabra reservada BEGIN. Puede utilizarse para declarar señales, tipos definidos por el usuario y constantes. También

```
ENTITY entity_name IS
    PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
           SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END entity_name ;
```

Figura A.2 Forma general de una declaración de entidad.

Tabla A.2 Los modos posibles para las señales que son puertos de entidad.

Modo	Propósito
IN	Utilizado para una señal que es una entrada a una entidad.
OUT	Utilizado para una señal que es una salida desde una entidad. El valor de la señal no puede usarse dentro de la entidad. Esto significa que en una instrucción de asignación, la señal puede aparecer sólo a la izquierda del operador <=.
INOUT	Utilizado para una señal que es tanto una entrada a una entidad como una salida desde la entidad.
BUFFER	Utilizado para una señal que es una salida desde una entidad. El valor de la señal puede usarse dentro de la entidad, lo cual significa que, en una instrucción de asignación, la señal puede aparecer tanto en el lado izquierdo como en el derecho del operador <=.

```
ARCHITECTURE architecture_name OF entity_name IS
    [SIGNAL declarations]
    [CONSTANT declarations]
    [TYPE declarations]
    [COMPONENT declarations]
    [ATTRIBUTE specifications]
BEGIN
    {COMPONENT instantiation statement ;}
    {CONCURRENT ASSIGNMENT statement ;}
    {PROCESS statement ;}
    {GENERATE statement ;}
END [architecture_name] ;
```

Figura A.3 Forma general de una arquitectura.

puede emplearse para declarar componentes y para especificar atributos; en las secciones A.6 y A.10.13 estudiamos las palabras reservadas COMPONENT y ATTRIBUTE, respectivamente.

La funcionalidad de la entidad se indica en el cuerpo de arquitectura, el cual va después de la palabra reservada BEGIN. Esta especificación comprende instrucciones que definen las funciones lógicas en el circuito, lo cual puede hacerse de diversas formas. En las secciones que siguen estudiaremos varias posibilidades.

Ejemplo A.9

En la figura A.4 se presenta el código de VHDL para una entidad llamada *fulladd,* que representa un circuito sumador completo. (El sumador completo se estudia en la sección 5.2.) La declaración de entidad especifica las señales de entrada y salida. El puerto de entrada *Cin* es el acarreo de entrada y los bits que se sumarán son los puertos de entrada *x* y *y.* Los puertos de salida son

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y   : IN     STD_LOGIC ;
           s, Cout     : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```

Figura A.4 Código para un sumador completo.

la suma, *s,* y el acarreo de salida, *Cout.* Las señales de entrada y salida se llaman *puertos* de la entidad. Este término se adoptó de la jerga de la electricidad, en la que un puerto es una conexión de entrada o de salida en un circuito eléctrico.

La arquitectura define el sumador completo por medio de ecuaciones lógicas. El nombre de la arquitectura puede ser cualquier nombre permitido en VHDL. Elegimos el nombre LogicFunc para este ejemplo simple. En términos de la forma general de la arquitectura de la figura A.3, una ecuación lógica es un tipo de asignación concurrente. Estas instrucciones se describen en la sección A.7.

A.5 PAQUETE

Un paquete de VHDL sirve como un depósito. Se utiliza para almacenar código de VHDL de uso general; por ejemplo, el código que define un tipo. El paquete puede incluirse para ser empleado por varios archivos de código fuente, los cuales pueden utilizar después las definiciones provistas en el paquete. Igual que una arquitectura, presentada en la sección A.4.2, un paquete puede tener dos partes principales: la *declaración del paquete* y el *cuerpo del paquete.* El cuerpo del paquete, *package_body,* es una parte opcional y no la usamos en este libro; un uso del cuerpo de un paquete es para definir funciones de VHDL, como las funciones de conversión presentadas en la sección A.2.13.

La forma general de la declaración de un paquete se presenta en la figura A.5. Las definiciones provistas en el paquete, como la definición de un tipo, pueden utilizarse en cualquier archivo de código fuente que incluya las instrucciones

```
LIBRARY library_name ;
USE library_name.package_name.all ;
```

```
PACKAGE package_name IS
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package_name ;
```

Figura A.5 Forma general de una declaración PACKAGE.

Library_name representa la ubicación del sistema de archivos de la computadora donde se almacena el paquete. Una biblioteca puede proporcionarse como parte de un sistema CAD, en cuyo caso se llama *biblioteca del sistema,* o ser creada por el usuario, caso en el que se denomina *biblioteca de usuario.* Un ejemplo de una biblioteca del sistema es la biblioteca *ieee*. En la sección A.2 estudiamos cuatro paquetes de esa biblioteca: *std_logic_1164, std_logic_signed, std_logic_unsigned* y *std_logic_arith.*

Un caso especial de una biblioteca de usuario se representa por medio del directorio del sistema de archivos donde se almacena el archivo de código fuente de VHDL que declara que un paquete está almacenado. Puede hacerse referencia a este directorio mediante el nombre de biblioteca *work,* que representa el *directorio de trabajo.* Por consiguiente, si se compila un archivo de código fuente que contiene una declaración de paquete llamada *user_package_name*, entonces el paquete puede usarse en otro archivo de código fuente (que se almacena en el mismo directorio del sistema de archivos) incluyendo las instrucciones

```
LIBRARY work ;
USE work.user_package_name.all ;
```

En realidad, para el caso especial de la biblioteca *work* no se requiere la cláusula LIBRARY, ya que la biblioteca de trabajo siempre está accesible.

En la figura A.5 se muestra que la declaración del paquete puede emplearse para declarar señales y componentes. Los componentes se estudian en la sección siguiente. Una señal declarada en un paquete puede ser usada por cualquier entidad de diseño que tenga acceso al paquete. Estas señales son similares en concepto a las variables globales utilizadas en los lenguajes de programación de computadora. Por el contrario, una señal declarada en una arquitectura puede usarse sólo dentro de esa arquitectura. Estas señales son análogas a las variables locales en un lenguaje de programación.

A.6 USO DE SUBCIRCUITOS

Una entidad de VHDL definida en un archivo de código fuente puede usarse como subcircuito en otro archivo de código fuente. En la jerga de VHDL el subcircuito se llama *componente.* Un subcircuito debe declararse mediante una *declaración de componente.* Esta instrucción especifica el nombre del subcircuito y proporciona los nombres de los puertos de entrada y salida. La declaración de componente puede aparecer ya sea en la región de una arquitectura o en una declaración de paquete. La forma general de la instrucción se muestra en la figura A.6. La sintaxis es parecida a la sintaxis de una declaración de entidad.

Una vez que se escribe una declaración de componente, éste puede *instanciarse* como un subcircuito, lo cual se hace usando una instrucción de *instanciación de componente*, que tiene la forma general

```
COMPONENT component_name
    [GENERIC ( parameter_name : integer := default_value {;
                parameter_name : integer := default_value} ) ;]
    PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
           SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END COMPONENT ;
```

Figura A.6 Forma general de una declaración de componente.

```
instance_name : component_name PORT MAP (
    formal_name => actual_name {, formal_name => actual_name} ) ;
```

Cada *formal_name* es el nombre de un puerto en el subcircuito. Cada *actual_name* es el nombre de una señal en el código que instancia el subcircuito. La sintaxis "formal_name =>" se proporciona de tal manera que el orden de las señales que viene después de las palabras reservadas PORT MAP no necesariamente tenga el mismo orden de los puertos en la declaración COMPONENT correspondiente. En la terminología de VHDL esto se llama *asociación por nombre.* Si los nombres de señal que van después de las palabras reservadas PORT MAP se escriben en el mismo orden que en la declaración COMPONENT, entonces "formal_name =>" no es necesario. Esto se denomina *asociación posicional.*

Un ejemplo que utiliza un componente (subcircuito) se muestra en la figura A.7. Este ejemplo proporciona el código para un sumador de acarreo en cascada de cuatro bits construido con cuatro instancias del subcircuito *fulladd*. Las entradas al sumador son el acarreo de entrada, *Cin,* y los 2 números de cuatro bits *X* y *Y.* La salida es la suma de cuatro bits, *S,* y el acarreo de salida, *Cout.* Hemos elegido el nombre Structure (Estructura) en la arquitectura porque el estilo jerárquico del código que usa subcircuitos se llama estilo *estructural*. Obsérvese que se declara una señal de tres bits, *C,* para representar los acarreos de salida de las etapas 0, 1 y 2. Esta señal se declara en la arquitectura, en vez de hacerlo en la declaración de entidad, pues se utiliza internamente en el circuito y no es un puerto de entrada o de salida.

La siguiente instrucción en la arquitectura provee la declaración de componente para el subcircuito *fulladd*. El cuerpo de arquitectura instancia cuatro copias del subcircuito sumador completo. En las primeras tres instrucciones de instanciación hemos usado la asociación posicional, ya que las señales se escriben en el mismo orden dado en la declaración para el componente fulladd de la figura A.4. La última instrucción de instanciación proporciona un ejemplo de asociación por nombre. Nótese que no está permitido usar el mismo nombre para una señal en la arquitectura que se usa para un nombre de puerto en un componente. Un ejemplo de esto es la señal *Cout*. Los nombres de señal utilizados en las instrucciones de instanciación especifican de manera implícita cómo están interconectadas las instancias del componente para crear la entidad adder.

Un segundo ejemplo de instanciación de componentes se muestra en la figura A.8. Un paquete llamado *lpm_components* en la biblioteca *lpm* se incluye en el código. Este paquete representa una colección de componentes llamada *Library of Parameterized Modules (LPM),* la cual es una biblioteca estandarizada de bloques de construcción de circuitos que en general son útiles para implementar circuitos lógicos.

El código de la figura A.8 instancia el componente de la LPM llamado *lpm_add_sub,* que se presentó en la sección 5.5.1. Este componente representa un circuito sumador/restador. La palabra reservada GENERIC sirve ahí para establecer el número de bits en el sumador/restador a 4. En la

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( Cin   : IN  STD_LOGIC ;
           X, Y  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN  STD_LOGIC ;
               s, Cout   : OUT STD_LOGIC) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin , X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP (
        x => X(3), y => Y(3), Cin => C(3), s => S(3), Cout => Cout ) ;
END Structure ;
```

Figura A.7 Código para un sumador de cuatro bits que utiliza la instanciación de componentes.

sección A.8 estudiamos las generalidades. La función de cada PORT en el componente *lpm_add_sub* es evidente a partir de los nombres de puerto utilizados en la instrucción de instanciación.

A.6.1 DECLARACIÓN DE UN COMPONENTE EN UN PAQUETE

En la figura A.5 se muestra que una declaración de componente puede darse en un paquete. Un ejemplo se muestra en la figura A.9. Define el paquete llamado *fulladd_package,* el cual proporciona la declaración de componente para la entidad *fulladd*. Este paquete puede almacenarse en un archivo de código fuente independiente o incluirse al final del archivo que define la entidad *fulladd* (véase la figura A.4). Cualquier código fuente que use la instrucción "USE work.fulladd_package.all" puede utilizar el componente *fulladd* como un subcircuito. En la figura A.10 se muestra cómo puede escribirse una entidad *adder* de acarreo en cascada de cuatro bits para usar el paquete. El código es el mismo que el de la figura A.7, excepto que incluye la cláusula USE adicional para el paquete y elimina la instrucción de declaración del componente desde la arquitectura.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY adderLPM IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adderLPM ;

ARCHITECTURE Structure OF adderLPM IS
BEGIN
    instance: lpm_add_sub
        GENERIC MAP ( LPM_WIDTH => 4)
        PORT MAP (
            dataa => X, datab => Y, Cin => Cin, result => S, Cout => Cout ) ;
END Structure ;
```

Figura A.8 Instanciación de un sumador de cuatro bits desde la biblioteca LPM.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figura A.9 Ejemplo de una declaración de paquete.

A.7 INSTRUCCIONES DE ASIGNACIÓN CONCURRENTE

Una instrucción de asignación concurrente sirve para asignar un valor a una señal en un cuerpo de arquitectura. En la figura A.4 se presentó un ejemplo, en el que las expresiones lógicas ilustran un tipo de instrucción de asignación concurrente. VHDL provee cuatro tipos de instrucciones de asignación concurrente: asignación de señal simple, asignación de señal seleccionada, asignación de señal condicional e instrucciones *generate*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figura A.10 Uso de un componente definido en un paquete.

A.7.1 ASIGNACIÓN DE SEÑAL SIMPLE

Una asignación de señal simple se utiliza para una expresión lógica o aritmética. La forma general es

```
signal_name <= expression ;
```

donde <= es el *operador de asignación* de VHDL. En los ejemplos siguientes se muestra cómo usarlo.

```
SIGNAL x1, x2, x3, f  : STD_LOGIC ;
        .
        .
        .
f <= (x1 AND x2) OR x3 ;
```

Esto define *f* en una expresión lógica, que implica cantidades de un solo bit. VHDL también soporta expresiones lógicas multibit, como en

```
SIGNAL A, B, C  : STD_LOGIC_VECTOR (1 TO 3) ;
        .
        .
        .
C  <=  A AND B ;
```

Esto da como resultado $C(1) = A(1) \cdot B(1)$, $C(2) = A(2) \cdot B(2)$, y $C(3) = A(3) \cdot B(3)$.

Un ejemplo de una expresión aritmética es

```
SIGNAL X, Y, S  : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
        .
        .
        .
S  <=  X + Y ;
```

Esta expresión representa un sumador de cuatro bits, sin acarreo de entrada ni de salida. Podemos declarar de manera opcional una señal de acarreo de entrada, *Cin,* y una señal de cinco bits, *Sum,* como sigue

```
SIGNAL  Cin  : STD_LOGIC ;
SIGNAL  Sum  : STD_LOGIC_VECTOR (4 DOWNTO 0) ;
```

Por tanto la instrucción

```
Sum  <=  ('0' & X) + Y + Cin ;
```

representa el sumador de cuatro bits con acarreo de entrada y de salida. Los cuatro bits de suma van de *Sum*(3) a *Sum*(0), mientras que el acarreo de salida es el bit *Sum*(4). La sintaxis ('0' & X) utiliza el *operador de concatenación* de VHDL, &, para poner un 0 en el extremo izquierdo de la señal *X.* El lector no debe confundir este uso del símbolo & con la operación lógica AND, que es el significado usual de este símbolo; en VHDL el AND lógico se indica mediante la palabra AND y & significa concatenar. La operación de concatenación antepone un dígito 0 en *X,* creando un número de cinco bits. VHDL requiere cuando menos que uno de los operandos de una expresión aritmética tenga el mismo número de bits que la señal utilizada para almacenar el resultado. El código completo para el sumador de cuatro bits con señales de acarreo se da en la figura A.11. Cabe observar que ésta es otra forma (en realidad, una mejor forma) de describir un sumador de cuatro bits, en comparación con el código estructural de la figura A.7. Obsérvese que la instrucción "S <= Sum(3 DOWNTO 0)" asigna los cuatro bits inferiores de la señal *Sum*, los cuales son los cuatro bits de suma, a la salida *S.*

A.7.2 ASIGNACIÓN DE LOS VALORES DE SEÑAL POR MEDIO DE OTHERS

Supóngase que se quiere establecer todos los bits de la señal *S* en 0. Como ya sabemos, una forma de hacerlo es escribir "S <= "0000" ;". Si el número de bits en *S* es grande, una manera más práctica de expresar la instrucción de asignación es por medio de la palabra reservada OTHERS, como en

```
S <= (OTHERS = > '0');
```

Esta instrucción también establece en 0 todos los bits de *S*, pero tiene la ventaja de funcionar para cualquier número de bits, no sólo cuatro. En general, el significado de (OTHERS => *Valor)* es establecer en *Value* cada bit del operando de destino. Un ejemplo del código que utiliza este constructor se muestra en la figura A.28.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Behavior OF adder IS
    SIGNAL Sum : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(3 DOWNTO 0) ;
    Cout <= Sum(4) ;
END Behavior ;
```

Figura A.11 Código para un sumador de cuatro bits, usando expresiones aritméticas.

A.7.3 ASIGNACIÓN DE SEÑAL SELECCIONADA

Una instrucción de asignación de señal seleccionada sirve para establecer el valor de una señal en una de varias posibilidades basadas en un criterio de selección. La forma general es

```
[label:] -- aquí puede colocarse una etiqueta adicional
WITH expression SELECT
    signal_name <= expression WHEN constant_value{,
                   expression WHEN constant_value} ;
```

Ejemplo A.10

```
SIGNAL x1, x2, Sel, f : STD_LOGIC ;
        .
        .
        .
WITH Sel SELECT
    f <=  x1 WHEN '0',
          x2 WHEN OTHERS ;
```

Este código describe un multiplexor dos a uno con *Sel* como la entrada select. En una asignación de señal seleccionada, todos los valores posibles de la entrada select, *Sel* en este caso, deben escribirse de manera explícita en el código. La palabra OTHERS proporciona una manera fácil de satisfacer este requisito. OTHERS representa todos los valores posibles que no están escritos.

En este caso los demás valores posibles son 1, Z, − y así por el estilo. Otro requisito para la asignación de señal seleccionada es que cada cláusula WHEN debe especificar un criterio que sea mutuamente excluyente del criterio de todas las demás cláusulas WHEN.

A.7.4 ASIGNACIÓN DE SEÑAL CONDICIONAL

Semejante a la asignación de señal seleccionada, la asignación de señal condicional se usa para establecer una señal en uno de varios valores posibles. La forma general es

```
[label:]
signal_name <=  expression WHEN logic_expression ELSE
                {expression WHEN logic_expression ELSE}
                expression ;
```

Un ejemplo es

```
f  <=  '1' WHEN x1 = x2 ELSE '0' ;
```

Cabe señalar una diferencia importante en comparación con la asignación de señal seleccionada. Las condiciones escritas después de cada cláusula WHEN no deben ser mutuamente excluyentes, ya que se da prioridad a las condiciones desde la primera hasta la última escritas. Esto se muestra en el ejemplo de la figura A.12. El código representa un codificador de prioridad en el que la solicitud de más alta prioridad se indica como la salida del circuito. (Los circuitos codificadores se describen en el capítulo 6.) La salida, *f*, del codificador de prioridad comprende dos bits cuyos valores dependen de las tres entradas, *req*1, *req*2 y *req*3. Si *req*1 es 1, entonces *f* se establece en 01. Si *req*2 es 1, entonces *f* se establece en 10, pero sólo si *req*1 no es también 1. Por consiguiente,

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY priority IS
    PORT ( req1, req2, req3  : IN   STD_LOGIC ;
           f                 : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    f <= "01" WHEN req1 = '1' ELSE
         "10" WHEN req2 = '1' ELSE
         "11" WHEN req3 = '1' ELSE
         "00" ;
END Behavior;
```

Figura A.12 Un codificador de prioridad descrito con una asignación de señal condicional.

```
generate_label:
FOR index_variable IN range GENERATE
        statement ;
        {statement ;}
END GENERATE ;


generate_label:
IF expression GENERATE
        statement ;
        {statement ;}
END GENERATE ;
```

Figura A.13 Formas generales de la instrucción GENERATE.

*req*1 tiene mayor prioridad que *req*2. De manera similar, *req*1 y *req*2 tienen mayor prioridad que *req*3. Por tanto, si *req*3 es 1, entonces *f* es 11, pero sólo si *req*1 y *req*2 no son también 1. Para este codificador de prioridad, si ninguna de las tres entradas es 1, entonces el valor 00 se asigna a *f*.

A.7.5 INSTRUCCIÓN GENERATE

Hay dos variantes de la instrucción GENERATE: FOR GENERATE e IF GENERATE. La forma general de los dos tipos se muestra en la figura A.13. La instrucción IF GENERATE pocas veces se necesita, pero FOR GENERATE se utiliza con frecuencia. Brinda una manera práctica de repetir ya sea una expresión lógica o una instanciación de componentes. En la figura A.14 se ilustra este último uso. El código de la figura equivale al código dado en la figura A.7.

A.8 DEFINICIÓN DE UNA ENTIDAD CON GENERIC

El código de la figura A.14 representa un sumador de números de cuatro bits. Es posible volver más general este código introduciendo un parámetro que represente el número de bits en el sumador. En la jerga de VHDL un parámetro como éste se llama GENERIC. En la figura A.15 se presenta el código para una entidad sumador de *n* bits, llamada *addern.* La palabra reservada GENERIC se emplea para definir el número de bits, *n,* que se va a añadir. Este parámetro se usa en el código, tanto en las definiciones de las señales *X, Y* y *S* como en la instrucción FOR GENERATE que instancia los *n* sumadores completos.

Es posible usar la función GENERIC con componentes que se instancian como subcircuitos en otro código. En la sección A.10.9 ofrecemos un ejemplo que utiliza la entidad *addern* como un subcircuito.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO 4) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO 3 GENERATE
        bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1)) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figura A.14 Ejemplo de una instanciación de componentes con FOR GENERATE.

A.9 INSTRUCCIONES DE ASIGNACIÓN SECUENCIALES

El orden en que aparecen las instrucciones de asignación concurrente no afecta el significado del código. Muchos tipos de circuitos lógicos pueden describirse con estas instrucciones. Sin embargo, VHDL también proporciona otro tipo de instrucciones, llamadas *de asignación secuencial,* para el que el orden sí afecta la semántica del código. Hay tres variantes de las instrucciones de asignación secuencial: instrucción IF, instrucción CASE e instrucciones LOOP.

A.9.1 INSTRUCCIÓN PROCESS

Como el orden en el que las instrucciones secuenciales aparecen en el código de VHDL es significativo, mientras que el orden de las instrucciones concurrentes no lo es, las primeras deben estar separadas de las segundas, lo cual se lleva a cabo con una instrucción PROCESS. La instrucción PROCESS aparece dentro de un cuerpo de arquitectura y encierra otras instrucciones. Las instrucciones IF, CASE y LOOP pueden aparecer sólo dentro de un proceso. La forma general de una instrucción de proceso (PROCESS) se muestra en la figura A.16. Su estructura es un tanto parecida a una arquitectura. Los objetos de datos VARIABLE pueden declararse (sólo)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO n-1 GENERATE
        stage: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1)) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figura A.15 Un sumador de *n* bits.

```
[process_label:]
PROCESS [( signal name {, signal name} )]
    [VARIABLE declarations]
BEGIN
    [WAIT statement]
    [Simple Signal Assignment Statements]
    [Variable Assignment Statements]
    [IF Statements]
    [CASE Statements]
    [LOOP Statements]
END PROCESS [process_label] ;
```

Figura A.16 Forma general de una instrucción PROCESS.

dentro del proceso. Cualquier variable declarada puede ser usada sólo por el código dentro del proceso; decimos que el *ámbito* de la variable está limitado al proceso. Para usar el valor de esta variable fuera del proceso, el valor de la variable puede asignarse a una señal. Los diversos elementos del proceso se explican mejor con algunos ejemplos, pero primero debemos explicar las instrucciones IF, CASE y LOOP.

Las instrucciones IF, CASE y LOOP pueden usarse para describir los circuitos combinacionales o los secuenciales. Las explicaremos mediante algunos ejemplos de circuitos combinacionales pues es más fácil comprenderlos. Los circuitos secuenciales se describen en la sección A.10.

A.9.2 INSTRUCCIÓN IF

La forma general de una instrucción IF se presenta en la figura A.17. Un ejemplo que utiliza una instrucción IF para lógica combinacional es

```
IF Sel = '0' THEN
    f <= x1 ;
ELSE
    f <= x2 ;
END IF ;
```

Este código define el multiplexor dos a uno usado como un ejemplo de una asignación de señal seleccionada en la sección anterior. En la sección A.10 se presentan ejemplos de lógica secuencial descritos con las instrucciones IF.

A.9.3 INSTRUCCIÓN CASE

La forma general de una instrucción CASE se muestra en la figura A.18. El *constant_value* puede ser un solo valor, como 2, una lista de valores separados por la barra |, como 2|3, o un rango, como 2 a 4. Un ejemplo de una instrucción CASE que se emplea para describir la lógica combinacional es

```
IF expression THEN
      statement ;
      {statement ;}
ELSIF expression THEN
      statement ;
      {statement ;}
ELSE
      statement ;
      {statement ;}
END IF ;
```

Figura A.17 Forma general de una instrucción IF.

```
CASE expression IS
      WHEN constant_value =>
            statement ;
            {statement ;}
      WHEN constant_value =>
            statement ;
            {statement ;}
      WHEN OTHERS =>
            statement ;
            {statement ;}
END CASE ;
```

Figura A.18 Forma general de una instrucción CASE.

```
CASE Sel IS
    WHEN '0' =>
          f <= x1 ;
    WHEN OTHERS =>
          f <= x2 ;
END CASE ;
```

Este código representa el mismo multiplexor dos a uno descrito en la sección A.9.2 que utiliza la instrucción IF. Semejante a una asignación de señal seleccionada, todas las combinaciones posibles de la expresión usada para las cláusulas WHEN deben escribirse; por consiguiente, se requiere la palabra reservada OTHERS. Además, todas las cláusulas WHEN en la instrucción CASE deben ser mutuamente excluyentes. Ejemplos de circuitos secuenciales descritos con la instrucción CASE se proporcionan en la sección A.10.10.

A.9.4 INSTRUCCIONES LOOP

VHDL provee dos tipos de instrucciones de ciclo: la instrucción FOR-LOOP y la instrucción WHILE-LOOP. Sus formas generales se muestran en la figura A.19. Estas instrucciones sirven para repetir una o más instrucciones de asignación secuencial de manera muy parecida a como se usa una instrucción FOR GENERATE a fin de repetir las instrucciones de asignación concurrentes. En la sección A.9.7 se presentan ejemplos de FOR-LOOP.

A.9.5 USO DE UN PROCESO PARA UN CIRCUITO COMBINACIONAL

Un ejemplo de una instrucción PROCESS se muestra en la figura A.20. Incluye el código para la instrucción IF de la sección A.9.2. Las señales *Sel, x*1 y *x*2 se muestran entre paréntesis después de la palabra reservada PROCESS. Indican de qué señales depende el proceso y se llaman *lista de sensibilidad* del proceso. Para un proceso que describe la lógica combinacional, como en este ejemplo, la lista de sensibilidad comprende todas las señales de entrada utilizadas dentro del proceso.

```
[loop_label:]
FOR variable_name IN range LOOP
      statement ;
      {statement ;}
END LOOP [loop_label] ;

[loop_label:]
WHILE boolean_expression LOOP
      statement ;
      {statement ;}
END LOOP [loop_label] ;
```

Figura A.19 Forma general de las instrucciones FOR-LOOP y WHILE-LOOP.

```
PROCESS ( Sel, x1, x2 )
BEGIN
      IF Sel = '0' THEN
            f <= x1 ;
      ELSE
            f <= x2 ;
      END IF ;
END PROCESS ;
```

Figura A.20 Una instrucción PROCESS.

En la jerga de VHDL un proceso se describe como sigue. Cuando el valor de una señal en la lista de sensibilidad cambia, el proceso se vuelve *activo.* Una vez activo, las instrucciones que contiene se "evalúan" en orden secuencial. Cualesquiera asignaciones de señal hechas en el proceso surten efecto sólo después que todas las instrucciones dentro de él se han evaluado. Decimos que las instrucciones de asignación de señal dentro del proceso están *previstas* y surtirán efecto al final del proceso.

El proceso describe un circuito lógico y se traduce en expresiones lógicas de la misma manera que las instrucciones de asignación concurrentes en un cuerpo de arquitectura. El hecho de que las instrucciones de proceso son evaluadas en secuencia provee una forma práctica de comprender la semántica del código dentro del proceso. En particular, un concepto clave es que si se hacen varias asignaciones a una señal dentro de un proceso, sólo la última que se evaluará tiene algún efecto. Esto se ilustra en el ejemplo siguiente.

A.9.6 ORDEN DE LAS INSTRUCCIONES

La instrucción IF de la figura A.20 describe un multiplexor que asigna cualquiera de dos entradas, *x*1 o *x*2, a la salida *f*. Otra forma de describir el multiplexor con una instrucción IF se muestra en la figura A.21. La instrucción "f <= x1 ;" se evalúa primero. No obstante, la señal *f* tal vez no cambia en realidad al valor de *x*1 porque puede haber una asignación subsiguiente a *f* en el código dentro de la instrucción process. En este punto del proceso, *x*1 representa el valor *predeterminado* para *f* si ninguna otra asignación a *f* se va a evaluar. Si suponemos que *Sel* = 1, entonces se evaluará la instrucción "f <= x2 ;". El efecto de esta segunda asignación a *f* es invalidar la asignación predeterminada. Por consiguiente, el resultado del proceso es que *f* se establece al valor de *x*2 cuando *Sel* = 1. Si suponemos que *Sel* = 0, entonces la condición IF fracasa y se asigna a *f* su valor predeterminado, *x*1.

Este ejemplo ilustra el efecto del orden de las instrucciones dentro de un proceso. Si el orden de las dos instrucciones se invirtiera, entonces la instrucción IF se evaluaría primero y la instrucción "f <= x1 ;" al último. Por tanto, el proceso siempre daría como resultado que *f* se estableciera en el valor de *x*1.

Memoria implícita

Considérese el proceso de la figura A.22. Es el mismo que el de la figura A.21 salvo que la instrucción de asignación predeterminada "f <= x1 ;" se ha eliminado. Como el proceso no especifica un valor predeterminado para *f* y no hay cláusula ELSE en la instrucción IF, el significado del proceso es que *f* debe conservar su valor actual cuando la condición IF no esté satisfecha.

```
PROCESS ( Sel, x1, x2 )
BEGIN
      f <= x1 ;
      IF Sel = 1 THEN
            f <= x2 ;
      END IF ;
END PROCESS ;
```

Figura A.21 Un ejemplo que ilustra el orden de las instrucciones dentro de PROCESS.

```
PROCESS ( Sel, x2 )
BEGIN
      IF Sel = 1 THEN
            f <= x2 ;
      END IF ;
END PROCESS ;
```

Figura A.22 Un ejemplo de memoria implícita.

La expresión siguiente se genera por medio del compilador de VHDL para este proceso

$$f = Sel \cdot x2 + \overline{Sel} \cdot f$$

Por tanto, cuando $Sel = 0$, el valor de $x2$ se "recuerda" en la salida f. En la terminología de VHDL esto se llama *memoria implícita.* Aunque rara vez es útil para circuitos combinacionales, mostraremos en breve que la memoria implícita es el concepto clave usado para describir circuitos secuenciales.

A.9.7 USO DE UNA VARIABLE EN UN PROCESO

Mencionamos antes que VHDL proporciona objetos de datos VARIABLE, además de objetos de datos SIGNAL. A diferencia de una señal, un objeto de datos variable no representa un cable en un circuito. Por ende, una variable puede usarse para describir la funcionalidad de un circuito lógico en formas que no son posibles utilizando una señal. Este concepto se ilustra en la figura A.23. El propósito del código es describir un circuito lógico que cuente el número de bits en la señal *X* de tres bits que son iguales a 1. El conteo se muestra usando la señal llamada *Count,* la cual es un entero sin signo de dos bits. Obsérvese que *Count* se declara con el modo *Buffer,* ya que se utiliza en el cuerpo de arquitectura de ambos lados, izquierdo y derecho, de un operador de asignación. En la tabla A.2 se explica el significado del modo *Buffer*.

Dentro del proceso, *Count* se establece inicialmente en 0. No se utilizan comillas para el número 0 en este caso, pues VHDL permite que un número decimal, el cual se indica sin comillas

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY numbits IS
    PORT ( X      : IN     STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : BUFFER INTEGER RANGE 0 TO 3 ) ;
END numbits ;

ARCHITECTURE Behavior OF numbits IS
BEGIN
    PROCESS ( X )   -- cuenta el número de bits en X con el valor 1
    BEGIN
        Count <= 0 ;   -- el 0 sin comillas es un número decimal
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END LOOP ;
    END PROCESS ;
END Behavior ;
```

Figura A.23 Una instrucción FOR-LOOP que no representa un circuito sensible.

(como mencionamos en la sección A.2.2) se asigne a una señal INTEGER. El código proporciona un FOR-LOOP con la variable de índice del ciclo *i*. Para los valores de *i* de 1 a 3, la instrucción IF dentro de FOR-LOOP verifica el valor del bit X(i); si es 1, entonces el valor de *Count* se incrementa. El código dado en la figura es código permisible de VHDL y puede compilarse sin generar ningún error. Sin embargo, no funcionará como se pretende, y no representa un circuito lógico sensible.

Hay dos razones por las que el código de la figura A.23 no funcionará como se desea. Primero, hay varias instrucciones de asignación para la señal *Count* dentro del proceso. Como se explicó para el ejemplo anterior, sólo la última de estas asignaciones surtirá efecto. Por consiguiente, si cualquier bit en *X* es 1, entonces la instrucción "Count <= '0';" no tendrá el efecto buscado de inicializar *Count* en 0, debido a que será invalidada por la instrucción de asignación en FOR-LOOP. Además, FOR-LOOP no funcionará como se desea, ya que cada iteración para la cual *X*(1) es 1 invalidará el efecto de la iteración anterior. La segunda razón por la que el código no es sensible consiste en que la instrucción "Count <= Count +'1';" describe un circuito con retroalimentación. Como el circuito es combinacional, esta retroalimentación dará como resultado oscilaciones y el circuito no será estable.

El comportamiento buscado del código de VHDL en la figura A.23 puede lograrse utilizando una variable en vez de una señal. Esto se ilustra en la figura A.24, en la que la variable *Tmp* se usa en lugar de la señal *Count* dentro del proceso. El valor de *Tmp* se asigna a *Count* al final del proceso. Obsérvese que las instrucciones de asignación para *Tmp* se indican con el operador := ,

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
    PORT ( X      : IN   STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : OUT  INTEGER RANGE 0 TO 3 ) ;
END Numbits ;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
    PROCESS ( X )   -- cuenta el número de bits en X iguales a 1
        VARIABLE Tmp : INTEGER ;
    BEGIN
        Tmp := 0 ;
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Tmp := Tmp + 1 ;
            END IF ;
        END LOOP ;
        Count <= Tmp ;
    END PROCESS ;
END Behavior ;
```

Figura A.24 Instrucción FOR-LOOP de la figura A.23 utilizando una variable.

a diferencia del operador <=. El operador := se denomina *operador de asignación de variable.* A diferencia de <=, no hace que la asignación sea *programada* hasta el final del proceso. La asignación de variables ocurre de inmediato. Esta asignación *inmediata* resuelve el primero de los dos problemas con el código de la figura A.23. El segundo problema también se resuelve usando una variable en vez de una señal. Como la variable no representa un cable en un circuito, FOR-LOOP no necesita interpretarse literalmente como un circuito con retroalimentación. Al usar la variable, la instrucción FOR-LOOP representa sólo un *comportamiento* deseado, o *funcionalidad,* del circuito. Cuando el código se traduce, el compilador de VHDL generará un circuito combinacional que implementa la funcionalidad expresada en esta instrucción FOR-LOOP.

Cuando el compilador de VHDL traduce el código de la figura A.24 produce el circuito con dos sumadores de dos bits mostrado en la figura A.25. Es posible ver cómo este circuito corresponde al FOR-LOOP en el código. El resultado de la primera iteración del ciclo es que *Count* se establece al valor de $X(1)$. Luego, la segunda iteración suma $X(1)$ a $X(2)$. Esto lo realiza el sumador superior de la figura. La tercera iteración suma $X(3)$ a la suma producida de la segunda iteración. Esto corresponde al sumador inferior. Cuando este circuito se mejora por los algoritmos de síntesis lógica, las expresiones resultantes para *Count* son

$$\text{Count}(1) = \text{X}(1)\text{X}(2) + \text{X}(1)\text{X}(3) + \text{X}(2)\text{X}(3)$$
$$\text{Count}(0) = \text{X}(1) \oplus \text{X}(2) \oplus \text{X}(3)$$

Estas expresiones representan un circuito sumador completo, con *Count*(0) como el resultado de la suma y *Count*(1) como el acarreo de salida. Resulta interesante observar que aun cuando el código de VHDL describe el comportamiento deseado del circuito de una manera abstracta,

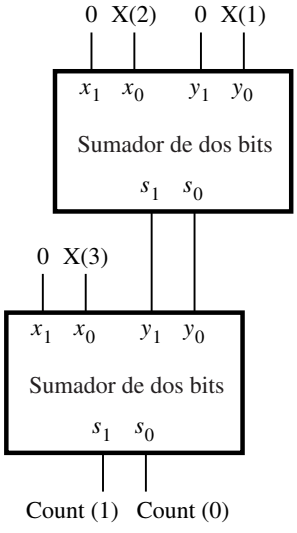


Figura A.25 El circuito generado a partir del código de la figura A.24.

usando un FOR-LOOP, en este ejemplo los algoritmos de síntesis lógica producen el circuito más eficaz, el cual es el sumador completo. Como dijimos al principio de este apéndice y en la sección 2.10, el estilo del código de la figura A.24 debe evitarse porque resulta difícil para el diseñador prever qué circuito lógico representa el código.

Como otro ejemplo del uso de una variable, en la figura A.26 se proporciona el código para una entidad de compuerta NAND de *n* bits, llamada *NANDn.* El número de entradas a la compuerta NAND se establece por medio del parámetro *n* de GENERIC. Las entradas son la señal *X* de *n* bits, y la salida es *f.* La variable *Tmp* se define en la arquitectura y originalmente se establece en el valor de la señal de entrada *X*(1). En el FOR LOOP, se aplica AND a *Tmp* de manera sucesiva con señales de entrada *X*(2) a *X*(*n*). Como *Tmp* es un objeto de datos variable, las asignaciones al mismo surten efecto de inmediato; no se programan para surtir efecto al final del proceso. El complemento de *Tmp* se asigna a *f*, con lo que la descripción de la operación NAND de *n* entradas se completa.

En la figura A.27 se muestra el mismo código presentado en la figura A.26 pero con el objeto de datos *Tmp* definido como una señal, en vez de como una variable. Este código da un resultado erróneo, ya que sólo la última instrucción incluida en el proceso surte algún efecto en *Tmp.* El código resulta en $Tmp = Tmp \cdot X(4)$, según lo determina la última iteración del FOR LOOP. Además, puesto que *Tmp* nunca se inicializa, su valor es desconocido. Por consiguiente, el valor de la salida $f = \overline{Tmp}$ se desconoce.

En la figura A.28 se muestra una manera de describir la compuerta NAND de *n* entradas por medio de señales. Aquí *Tmp* se define como una señal de entrada de *n* bits, la cual se esta-

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
BEGIN
    PROCESS ( X )
        VARIABLE Tmp : STD_LOGIC ;
    BEGIN
        Tmp := X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp := Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figura A.26 Uso de una variable para describir una compuerta NAND de *n* entradas.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC ;
BEGIN
    PROCESS ( X )
    BEGIN
        Tmp <= X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp <= Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figura A.27 El código de la figura A.26 usando una señal.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC_VECTOR(1 TO n) ;
BEGIN
    Tmp <= (OTHERS => '1') ;
    f <= '0' WHEN X = Tmp ELSE '1' ;
END Behavior ;
```

Figura A.28 Uso de una señal para describir una compuerta NAND de *n* entradas.

blece para contener *n* unos usando el constructor (OTHERS => '1'). La asignación de señal condicional especifica que *f* es 0 sólo si todos los bits en la entrada *X* son 1*;* por tanto, describe la operación NAND.

Un ejemplo final de variables utilizadas en un circuito secuencial se presenta en la sección A.10.7. En general, el uso tanto de las variables como de las señales del código de VHDL puede causar confusión porque implican una semántica distinta. Como las variables no necesariamente representan cables en un circuito, el significado del código que emplea variables a veces está mal definido. Para evitar confusión, en esta obra usamos variables sólo para los índices de los ciclos en las instrucciones FOR GENERATE y FOR LOOP. Excepto para propósitos similares, el lector debe evitar utilizar variables, pues no se precisan para describir circuitos lógicos.

A.10 CIRCUITOS SECUENCIALES

Aun cuando los circuitos combinacionales pueden describirse mediante instrucciones de asignación ya sean concurrentes o secuenciales, los circuitos secuenciales sólo pueden describirse con instrucciones de asignación secuencial. Ahora presentaremos algunos ejemplos representativos de circuitos secuenciales.

A.10.1 UN LATCH D ASÍNCRONO

En la figura A.29 se presenta el código para un latch D asíncrono. La lista de sensibilidad del proceso incluye tanto la entrada de datos del latch, *D,* como el reloj, *clk.* Por tanto, siempre que ocurra un cambio en el valor, ya sea de *D* o de *clk,* el proceso se vuelve activo. La instrucción

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, clk  : IN     STD_LOGIC ;
           Q       : OUT  STD_LOGIC ) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, clk )
    BEGIN
        IF clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.29 Un latch D asíncrono.

IF especifica que Q debe establecerse en el valor de *D* siempre que el reloj sea 1. No hay una cláusula ELSE en la instrucción IF. Como explicamos para la figura A.22, esto implica que Q debe conservar su valor actual cuando la condición IF no se cumpla.

A.10.2 FLIP-FLOP D

En la figura A.30 se muestra un proceso que es ligeramente distinto al de la figura A.29. La lista de sensibilidad incluye sólo la señal *Clock*, lo que significa que el proceso sólo está activo cuando el valor de *Clock* cambia. La condición de la instrucción IF parece poco común. La sintaxis Clock'EVENT representa un *cambio* en el valor de la señal de reloj (*Clock*). En la jerga de VHDL, 'EVENT se conoce como un *atributo,* y la combinación de 'EVENT con un nombre de señal, como *Clock,* produce una condición lógica. La combinación en la instrucción IF de las dos condiciones Clock'EVENT y Clock = '1' especifica que Q debe asignarse al valor de *D* cuando "un cambio ocurre en el valor de *Clock,* y *Clock* ahora es 1". Esto describe una transición de la señal de nivel bajo a alto; por ende, el código describe un flip-flop D disparado por el flanco positivo.

El paquete *std_logic_1164* define las dos funciones llamadas *rising_edge* y *falling_edge.* Pueden usarse como una notación abreviada para la condición que revisa la ocurrencia de un flanco del reloj. En la figura A.30 podemos remplazar la línea "IF Clock'EVENT AND Clock = '1' THEN" con la línea equivalente "IF rising_edge(Clock) THEN". No usamos *rising_edge* o *failing_edge* en este libro; se mencionan por conocimiento general.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.30 Flip-flop D.

A.10.3 USO DE UNA INSTRUCCIÓN WAIT UNTIL

El proceso de la figura A.31 utiliza una sintaxis diferente para describir un flip-flop D. La sincronización con el flanco del reloj se indica por medio de la instrucción "WAIT UNTIL Clock'EVENT AND Clock = 1 ;". Un proceso que usa una instrucción WAIT UNTIL es un caso especial debido a que se omite la lista de sensibilidad. El empleo de esta instrucción WAIT UNTIL especifica de manera implícita que la lista de sensibilidad sólo incluye *Clock.* Para nuestros propósitos, que consisten en utilizar VHDL para la síntesis de circuitos, un proceso puede incluir una instrucción WAIT UNTIL sólo si es la primera del proceso.

La instrucción WAIT UNTIL anterior puede escribirse de manera más simple como

```
WAIT UNTIL Clock = '1' ;
```

que significa "esperar el siguiente flanco positivo del reloj de la señal *Clock*". Pero como algunas herramientas de síntesis CAD requieren la inclusión del atributo 'EVENT, incluimos el atributo en nuestros ejemplos.

Como vimos en las figuras A.30 y A.31, tanto las instrucciones IF como WAIT UNTIL pueden utilizarse para describir flip-flops. Si un proceso sólo define flip-flops, entonces no importa cuál constructor se emplee. Sin embargo, en los diseños prácticos un proceso a menudo comprende muchas instrucciones. Si una o más de ellas indica un subcircuito combinacional, entonces es preciso usar las instrucciones IF en los flip-flops internos cuando se desee. Si se ocupa la instrucción WAIT UNTIL, la cual debe ser la primera instrucción del proceso, entonces habrá flip-flops inferidos para todas las instrucciones del proceso. Por ello los diseñadores prefieren usar la instrucción IF.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figura A.31 Código equivalente al de la figura A.30, que utiliza una instrucción WAIT UNTIL.

A.10.4 UN FLIP-FLOP CON *RESET* ASÍNCRONO

En la figura A.32 se presenta un proceso similar al de la figura A.30. Describe un flip-flop D con una entrada *reset*, o de borrado, asíncrono. La señal *reset* tiene el nombre de *Resetn.* Cuando *Resetn* = 0, la salida del flip-flop Q se establece en 0. Añadir la letra *n* al nombre de una señal es una convención muy útil para indicar una señal activa en nivel bajo.

A.10.5 *RESET* SÍNCRONO

En la figura A.33 se muestra cómo un flip-flop con una entrada *reset* síncrona puede describirse con la instrucción IF. En la figura A.34 se presenta una especificación basada en la instrucción WAIT UNTIL.

A.10.6 REGISTROS

Un método posible para describir un registro multibit es crear una entidad que instancie varios flip-flops. Un método más práctico se ilustra en la figura A.35. Presenta el mismo código mostrado en la figura A.32 pero usando la entrada *D* de cuatro bits de STD_LOGIC_VECTOR y la salida Q de cuatro bits. El código describe un registro de cuatro bits de borrado asíncrono.

En la figura A.36 aparece el código para una entidad llamada *regn.* Muestra cómo el código de la figura A.35 puede extenderse para representar un registro de *n* bits. El número de flip-flops se establece por medio del parámetro genérico *n.*

El código de la figura A.37 muestra cómo una entrada *enable* puede añadirse al registro de *n* bits de la figura A.36. Cuando el flanco activo del reloj ocurre, los flip-flops en el registro no

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.32 Flip-flop D con *reset* asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            IF Resetn = '0' THEN
                Q <= '0' ;
            ELSE
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.33 Flip-flop D con reset síncrono, que utiliza una instrucción IF.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.34 Flip-flop D con reset síncrono, que utiliza una instrucción WAIT UNTIL.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg4 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "0000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.35 Código para un registro de cuatro bits con borrado asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.36 Código para un registro de *n* bits con borrado asíncrono.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D          : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Resetn     : IN   STD_LOGIC ;
           E, Clock   : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figura A.37 Código de VHDL para un registro de *n* bits con una entrada enable.

pueden cambiar sus valores almacenados si la entrada enable E es 0. Si $E = 1$, el registro responde al flanco activo del reloj de manera normal.

A.10.7 REGISTROS DE CORRIMIENTO

Un ejemplo de código que define un registro de corrimiento de cuatro bits se muestra en la figura A.38. Las líneas de código se numeran para facilitar la referencia. El registro de corrimiento tiene una entrada serial, *w,* y salidas paralelas, Q. El bit en el extremo derecho del registro es Q(4) y el del extremo izquierdo es Q(1); el corrimiento se realiza de derecha a izquierda. La arquitectura declara la señal *Sreg,* la cual se utiliza para describir la operación de desplazamiento. Todas las asignaciones a *Sreg* están sincronizadas con el flanco activo del reloj por medio de la condición IF; por consiguiente, *Sreg* representa las salidas de los flip-flops. La instrucción en la línea 13 especifica que el valor de *w* se asigna a *Sreg*(4). Como explicamos antes, esta asignación no surte efecto de inmediato sino que se programa para que ocurra al final del proceso. En la línea 14 el valor actual de *Sreg*(4), antes que éste se desplace como resultado de la línea 13, se asigna a *Sreg*(3). Las líneas 15 y 16 completan la operación de corrimiento. Asignan los valores actuales de *Sreg*(3) y *Sreg*(2), antes que cambien como resultado de las líneas 14 y 15, a *Sreg*(2) y *Sreg*(1), respectivamente. Por último, *Sreg* se asigna a las salidas Q.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN   STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8       SIGNAL Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
9   BEGIN
10      PROCESS ( Clock )
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) <= w ;
14              Sreg(3) <= Sreg(4) ;
15              Sreg(2) <= Sreg(3) ;
16              Sreg(1) <= Sreg(2) ;
17          END IF ;
18      END PROCESS ;
19      Q <= Sreg ;
20  END Behavior ;
```

Figura A.38 Código para el registro de corrimiento de cuatro bits.

El punto clave que debe advertirse en el código de la figura A.38 es que las instrucciones de asignación en las líneas 13 a 16 no surten efecto hasta el final del proceso. Por consiguiente, todos los flip-flops cambian sus valores al mismo tiempo, según se requirió en el registro de corrimiento. Podríamos escribir las instrucciones de las líneas 13 a 16 en cualquier orden sin cambiar el significado del código.

En la sección A.9.7 expusimos las variables y mostramos cómo difieren de las señales. Como otro ejemplo de la semántica que supone el uso de las variables, en la figura A.39 aparece el código de la figura A.38 pero con *Sreg* declarado como una variable, en vez de como una señal. La instrucción de la línea 13 asigna el valor de *w* a *Sreg*(4). Puesto que *Sreg* es una variable, la asignación surte efecto de inmediato. En la línea 14 el valor de *Sreg*(4), que ya ha cambiado a *w*, se asigna a *Sreg*(3). Por tanto, la línea 14 hace que *Sreg*(3) = *w.* De manera similar, las líneas 15 y 16 establecen *Sreg*(2) y *Sreg*(1) en el valor de *w.* El código no describe el registro de corrimiento buscado, sino que carga todos los flip-flops con el valor de la entrada *w*.

Para que el código de la figura A.39 describa correctamente un registro de corrimiento, el orden de las líneas 13 a 16 debe invertirse. Por ende, la primera asignación establece *Sreg*(1) en el valor de *Sreg*(2), la segunda establece *Sreg*(2) en el valor de *Sreg*(3), y así sucesivamente. Cada asignación sucesiva no se ve afectada por la asignación que la precede; por consiguiente, la semántica del uso de variables no ocasiona un problema. Como dijimos en la sección A.9.7, puede ser confuso utilizar ambas señales y variables al mismo tiempo, ya que implican una semántica distinta.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN     STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8   BEGIN
9       PROCESS ( Clock )
10          VARIABLE Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) := w ;
14              Sreg(3) := Sreg(4) ;
15              Sreg(2) := Sreg(3) ;
16              Sreg(1) := Sreg(2) ;
17          END IF ;
18          Q <= Sreg ;
19      END PROCESS ;
20  END Behavior ;
```

Figura A.39 El código de la figura A.38, pero usando una variable.

A.10.8 CONTADORES

En la figura A.40 se muestra el código para un contador de cuatro bits con una entrada reset asíncrona. El contador también tiene una entrada enable, *E*. En el flanco positivo del reloj, si *E* es 1, el conteo se incrementa. Si $E = 0$, el contador mantiene su valor actual. Como los contadores suelen ser necesarios en los circuitos lógicos, casi todos los sistemas CAD proporcionan una selección de ellos que puede instanciarse en un diseño.

A.10.9 USO DE SUBCIRCUITOS CON PARÁMETROS GENERIC

Hemos mostrado varios ejemplos de entidades de VHDL que incluyen parámetros generic. Cuando estos subcircuitos se usan como componentes en otro código, los parámetros generic pueden establecerse a cualesquiera valores que se necesiten. Para dar un ejemplo de una instanciación de componentes que emplea parámetros generic, considérese el circuito mostrado en la figura A.41. El circuito suma el número binario representado por la entrada *X* de *k* bits a sí mismo cierto número de veces. Un circuito como éste se llama *acumulador*. Para almacenar el

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY count4 IS
    PORT ( Resetn    : IN    STD_LOGIC ;
           E, Clock  : IN    STD_LOGIC ;
           Q         : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
END count4 ;

ARCHITECTURE Behavior OF count4 IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figura A.40 Ejemplo de contador.

resultado de cada operación de suma, el circuito incluye un registro de *k* bits. El registro tiene una entrada reset asíncrona, *Resetn.* También tiene una entrada enable, *E,* la cual se controla mediante un contador de cuatro bits. El contador tiene una entrada clear asíncrona y una entrada enable de conteo. El circuito opera al borrar primero todos los bits del registro y establecer el contador en 0. Luego, en cada ciclo de reloj, el contador se incrementa y los resultados de la suma desde el sumador se almacenan en el registro. Cuando el contador llega al valor 1111, la compuerta NAND establece en 0 las entradas enable tanto del registro como del contador. Por consiguiente, el circuito permanece en ese estado hasta que se inicializa de nuevo. El valor final almacenado en el registro es igual a 15*X.*

Podemos representar el circuito acumulador utilizando varios subcircuitos descritos en el apéndice: *addern* (figura A.15), *NANDn* (figura A.28), *regne* y *count4.* Colocamos las instrucciones de declaración de componente para todos estos subcircuitos en un paquete, llamado *components,* el cual se muestra en la figura A.42.

El código completo para el acumulador se presenta en la figura A.43. Este código utiliza el parámetro generic *k* para representar el número de bits en la entrada *X.* El uso de este parámetro en el código facilita el cambio del ancho de bits posteriormente si así se desea. La arquitectura define la señal *Sum* para representar las salidas del sumador; por simplicidad, ignoramos la posibilidad de un desbordamiento aritmético y suponemos que la suma puede representarse

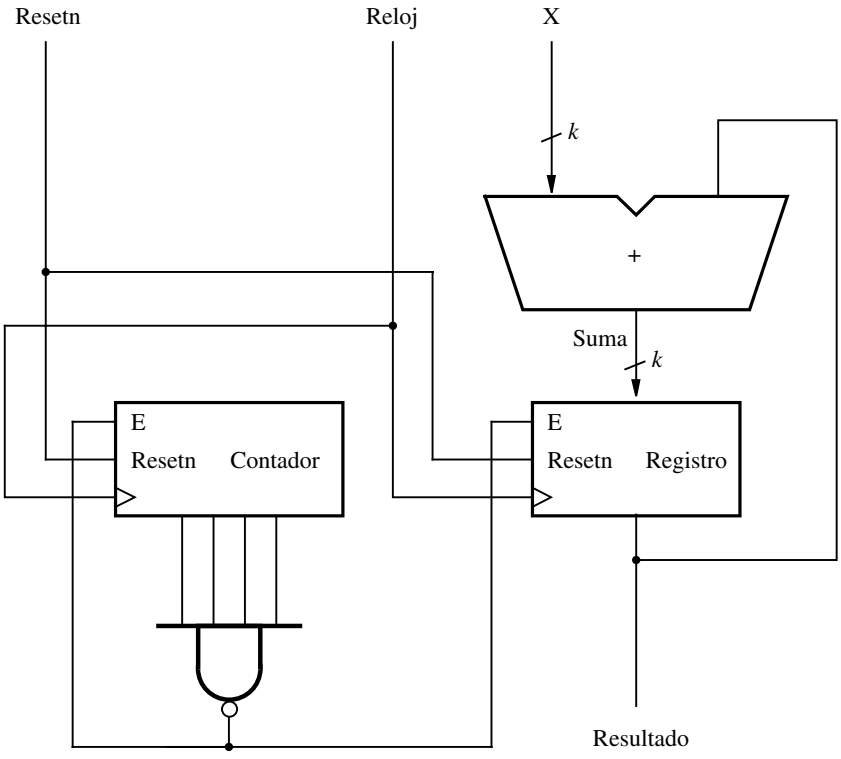


Figura A.41 El circuito acumulador.

utilizando *k* bits. La señal *C* de cuatro bits representa las salidas desde el contador. La señal *Stop* se conecta a las entradas enable en el registro y en el contador.

La instrucción etiquetada *adder* instancia el subcircuito *addern*. Las palabras reservadas GENERIC MAP se usan para especificar el valor del parámetro generic del sumador, *n.* La sintaxis (n => k) establece el número de bits en el sumador a *k.* No necesitamos el puerto de acarreo de entrada en el sumador, pero una señal debe conectarse a él. La señal *Zero_bit,* la cual se establece en '0' en el código, se utiliza para el puerto de acarreo de entrada (la sintaxis de VHDL no permite que un valor constante, como '1', se asocie directamente con un puerto; por consiguiente debe definirse una señal para este propósito). Las entradas de datos de *k* bits al sumador son *X* y la salida del registro se llama *Result.* La salida de la suma desde el sumador se denomina *Sum,* y el acarreo de salida, el cual no se utiliza en el circuito, se llama *Cout.*

El subcircuito *regne* se instancia en la instrucción etiquetada *reg.* GENERIC MAP se emplea para establecer el número de bits en el registro a *k.* La entrada del registro de *k* bits es proporcionada por la salida *Sum* desde el sumador. La salida del registro se llama *Result;* esta señal representa la salida del circuito acumulador. Tiene el modo BUFFER en la declaración de entidad. Esto es necesario en la sintaxis de VHDL para que la señal se conecte a un puerto en un componente instanciado.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    COMPONENT addern - - sumador de n bits
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( Cin    : IN    STD_LOGIC ;
               X, Y   : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               S      : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Cout   : OUT   STD_LOGIC ) ;
    END COMPONENT ;

    COMPONENT regne - - registro de n bits con enable
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( D          : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Resetn     : IN    STD_LOGIC ;
               E, Clock   : IN    STD_LOGIC ;
               Q          : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT count4 - - contador de 4 bits con enable
        PORT ( Resetn     : IN    STD_LOGIC ;
               E, Clock   : IN    STD_LOGIC ;
               Q          : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT NANDn - - compuerta AND de n bits
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( X   : IN    STD_LOGIC_VECTOR(1 TO n) ;
               f   : OUT   STD_LOGIC ) ;
    END COMPONENT ;

END components ;
```

Figura A.42 Declaraciones de componente para el circuito acumulador.

Los componentes *count4* y *NANDn* se instancian en las instrucciones etiquetadas *Counter* y *NANDgate.* No tenemos que usar las palabras reservadas GENERIC MAP para *NANDn,* ya que el valor predeterminado de este parámetro genérico es 4, que es el valor requerido en esta aplicación.

A.10.10 UNA MÁQUINA DE ESTADO FINITITO TIPO MOORE

En la figura A.44 se muestra el diagrama de estado de una máquina Moore simple. El código para esta máquina aparece en la figura A.45. La señal llamada *y* representa el estado de la máquina. Se declara con un tipo enumerado, *State_Type* que tiene los tres valores posibles A, B y C.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY accum IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( Resetn, Clock  : IN      STD_LOGIC ;
           X              : IN      STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           Result         : BUFFER  STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END accum ;

ARCHITECTURE Structure OF accum IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL Zero_bit, Cout, Stop : STD_LOGIC ;
BEGIN
    Zero_bit <= '0' ;
    adder: addern
        GENERIC MAP ( n => k )
        PORT MAP ( Zero_bit, X, Result, Sum, Cout ) ;
    reg: regne
        GENERIC MAP ( n => k )
        PORT MAP ( Sum, Resetn, Stop, Clock, Result ) ;
    Counter: count4
        PORT MAP ( Clock, Resetn, Stop, C ) ;
    NANDgate: NANDn
        PORT MAP ( C, Stop ) ;
END Structure ;
```

Figura A.43 Código para el circuito acumulador.

Cuando el código se compila, el compilador de VHDL realiza automáticamente una asignación de estado para seleccionar los patrones de bits apropiados para los tres estados. El comportamiento de la máquina se define por medio del proceso con la lista de sensibilidad que comprende las señales reset y clock.

El código de VHDL incluye una entrada reset asíncrona que pone la máquina en el estado A. La tabla de estado para la máquina se define con una instrucción CASE. Cada cláusula WHEN corresponde a un estado actual de la máquina, y la instrucción IF dentro de la cláusula WHEN especifica el estado siguiente al que se llegará después del siguiente flanco positivo de la señal de reloj. Como se trata de una máquina tipo Moore, la salida z puede definirse como una instrucción de asignación concurrente separada que sólo depende del estado actual de la máquina. Otra posibilidad es que el valor apropiado para z podría haberse especificado dentro de cada cláusula WHEN de la instrucción CASE.

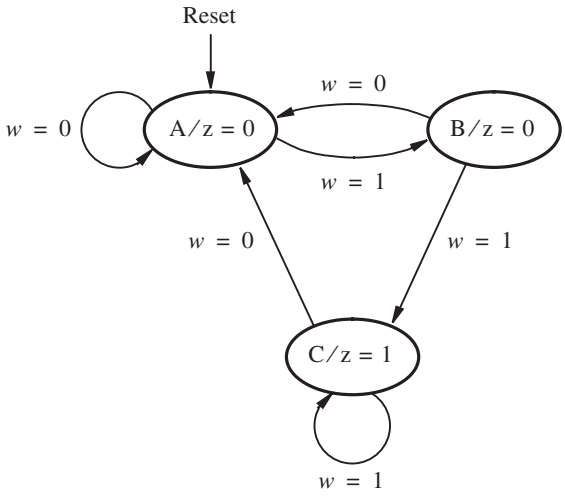


Figura A.44 Diagrama de estado de una FSM simple tipo Moore.

Otra forma de describir una máquina de estado finito tipo Moore se proporciona en la arquitectura de la figura A.46. Dos señales se utilizan para describir cómo se mueve la máquina de un estado a otro. La señal *y_present* representa las salidas de los flip-flops de estado y la señal *y_next* representa las entradas de los flip-flops. El código tiene dos procesos. El proceso superior describe un circuito combinacional. Emplea una instrucción CASE para especificar los valores que *y_next* debe tener por cada valor de *y_present.* El otro proceso representa un circuito secuencial, el cual especifica que el valor de *y_next* se asigna a *y_present* en el flanco positivo del reloj. El proceso también indica que *y_present* debe tomar el valor de A cuando *Resetn* es 0, lo cual proporciona el reset asíncrono.

A.10.11 UNA MÁQUINA DE ESTADO FINITITO TIPO MEALY

En la figura A.47 se muestra un diagrama de estado para una máquina tipo Mealy simple. El código correspondiente aparece en la figura A.48. El código es el mismo que el de la figura A.45 excepto porque la salida *z* se especifica usando una instrucción CASE separada. La instrucción CASE establece que cuando la FSM se encuentra en el estado *A, z* debe ser 0, pero cuando se halla en el estado *B, z* debe tomar el valor de *w.* Esta instrucción CASE describe bien la lógica necesaria para *z*. Sin embargo, no está claro por qué debemos utilizar una segunda instrucción CASE en vez de especificar el valor de *z* dentro de la instrucción CASE que define la tabla de estado para la máquina. Este método no funcionaría correctamente porque la instrucción CASE para la tabla de estado está anidada dentro de la instrucción IF que espera que ocurra un flanco de reloj. Por consiguiente, si colocamos el código para *z* dentro de esta instrucción CASE, entonces el valor de *z* sólo podría cambiar como resultado de un flanco de reloj, lo que no satisface los requisitos de la FSM tipo Mealy, pues el valor de *z* no sólo depende del estado de la máquina sino también del valor de la entrada *w.*

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY moore IS
    PORT ( Clock   : IN   STD_LOGIC ;
           w       : IN   STD_LOGIC ;
           Resetn  : IN   STD_LOGIC ;
           z       : OUT  STD_LOGIC ) ;
END moore ;

ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

Figura A.45 Un ejemplo de una máquina de estado finito tipo Moore.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figura A.46 Código equivalente al de la figura A.45, pero que utiliza dos procesos.

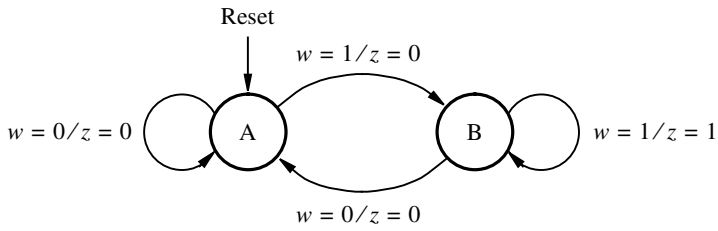


Figura A.47 Diagrama de estado para una FSM tipo Mealy.

A.11 ERRORES COMUNES EN EL CÓDIGO DE VHDL

En esta sección se enumeran algunos de los errores comunes que nuestros estudiantes cometen cuando escriben código de VHDL.

Nombres de ENTITY y ARCHITECTURE

Los nombres usados en una declaración de entidad (ENTITY) y la arquitectura (ARCHITECTURE) correspondiente deben ser idénticos. El código

```
ENTITY adder IS
⋮
END adder ;
ARCHITECTURE Structure OF adder4 IS
⋮
END Structure ;
```

es erróneo porque la declaración ENTITY utiliza el nombre *adder,* mientras que la arquitectura utiliza el nombre *adder4.*

Punto y coma faltante

Cada instrucción de VHDL debe terminar con un punto y coma.

Uso de apóstrofos y comillas

Los apóstrofos se utilizan para datos de un solo bit, mientras que las comillas se emplean para datos multibit; los datos enteros se escriben sin comillas. En la sección A.2 se dan algunos ejemplos.

Instrucciones combinacionales en comparación con las secuenciales

Las instrucciones combinacionales incluyen asignaciones de señal simple, asignaciones de señal seleccionada e instrucciones generate. Las asignaciones de señal simple pueden utilizarse ya sea fuera o dentro de una instrucción PROCESS. Los otros tipos de instrucciones combinacionales sólo pueden usarse fuera de una instrucción PROCESS.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn  : IN   STD_LOGIC ;
           w              : IN   STD_LOGIC ;
           z              : OUT  STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figura A.48 Ejemplo de una máquina tipo Mealy.

Las instrucciones secuenciales incluyen las instrucciones IF, CASE y LOOP. Cada uno de estos tipos de instrucciones sólo puede emplearse dentro de una instrucción de proceso (process).

Instanciación de componentes

La instrucción siguiente contiene dos errores

```
control: shiftr GENERIC MAP ( K => 3 ) ;
                PORT MAP ( '1', Clock, w, Q ) ;
```

No debe haber un punto y coma al final de la primera línea, pues las dos líneas representan una sola instrucción de VHDL. Además, es ilegal asociar un valor constante ('1') a un puerto en un componente. El código que aparece a continuación muestra cómo pueden corregirse los dos errores.

```
SIGNAL High ;
.
.
.
High <= '1' ;
control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( High, Clock, w, Q ) ;
```

Nombres de etiquetas, señales y variables

Es ilegal utilizar cualquier palabra reservada de VHDL como un nombre de señal, etiqueta o variable. Por ejemplo, es ilícito llamar *In* o *Out* a una señal. También es ilegal utilizar el mismo nombre varias veces para cualquier etiqueta, señal o variable en un diseño de VHDL. Un error común consiste en emplear el mismo nombre para una señal y una variable que se usa como el índice en una instrucción generate o loop. Por ejemplo, si el código recurre a la instrucción generate

```
Generate_label:
FOR i IN 0 TO 3 GENERATE
    bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
END GENERATE ;
```

entonces es ilegal definir una señal llamada *i* (o *I*, porque VHDL no distingue mayúsculas de minúsculas).

Memoria implícita

Como se muestra en la sección A.10, la memoria implícita se utiliza para describir elementos de almacenamiento. Ha de tenerse cuidado para evitar memoria implícita no deseada. El código

```
IF LA = '1' THEN
    EA <= '1' ;
END IF ;
```

da como resultado memoria implícita para la señal *EA*. Si no es lo que se busca, entonces el código puede corregirse al escribir

```
IF LA = '1' THEN
    EA <= '1' ;
ELSE
    EA <= '0' ;
END IF ;
```

La memoria implícita también se aplica a las instrucciones CASE. La instrucción

```
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

no especifica el valor de la señal *EA* cuando *y* no es igual a *S*1, y no especifica el valor de *EB* cuando *y* no es igual a *S2.* A fin de evitar la memoria implícita tanto para *EA* como para *EB* debe asignarse a estas señales valores predeterminados, como en el código

```
EA <= '0' ; EB <= '0' ;
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

En general, el diseñador debe intentar escribir código de VHDL que contenga el menor número de errores posible, puesto que encontrar el origen de un error suele ser difícil.

A.12 COMENTARIOS FINALES

En este apéndice se describen todos los constructores importantes de VHDL que sirven para la síntesis de los circuitos lógicos. Como mencionamos, no estudiamos ninguna función de VHDL que sea útil para la simulación de circuitos o para otros propósitos. Un lector que desee aprender más de VHDL puede remitirse a libros especializados [1-7].

BIBLIOGRAFÍA

1. Institute of Electrical and Electronics Engineers, "1076-1993 IEEE Standard VHDL Language Reference Manual", 1993.
2. D. L. Perry, *VHDL,* 3a. ed. (McGraw-Hill: Nueva York, 1998).
3. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2a. ed. (McGraw-Hill: Nueva York, 1998).
4. J. Bhasker, *A VHDL Primer*, 3a. ed. (Prentice-Hall: Englewood Cliffs, NJ, 1998).
5. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
6. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
7. S. Yalamanchili, *VHDL Starter's Guide* (Prentice-Hall: Upper Saddle River, NJ, 1998).

apéndice

B

TUTORIAL 1
USO DEL SOFTWARE CAD QUARTUS II

Quartus II es un moderno sistema CAD. Como la mayor parte de las herramientas CAD, se mejora y actualiza continuamente, por lo que se han liberado varias versiones. La conocida como Quartus II 4.0 se incluye en el CD-ROM adjunto a este libro. Por simplicidad, en nuestro estudio nos referiremos a este paquete simplemente como Quartus II.

En este tutorial exponemos el diseño de circuitos lógicos utilizando Quartus II. Presentamos instrucciones paso a paso para realizar la captura del diseño con dos métodos: el uso de la captura esquemática y la escritura de código de VHDL, así como con una combinación de ambos. El tutorial también ilustra el proceso de simulación.

B.1 INTRODUCCIÓN

En este tutorial se presupone que el lector tiene acceso a una computadora con Quartus II instalado. Las instrucciones de instalación se proporcionan con el software. Quartus II se ejecuta en varios tipos de sistemas de cómputo. Para este tutorial presuponemos que el lector tiene una computadora con un sistema operativo de Microsoft (Windows NT, Windows 2000 o Windows XP). Aun cuando Quartus II opera igual en todos los tipos de computadoras que lo soportan, hay diferencias menores. Un lector que no utilice un sistema operativo Windows de Microsoft puede experimentar ligeras discrepancias con este tutorial. Ejemplos de diferencias potenciales son la ubicación de los archivos en el sistema de archivos de la computadora y la apariencia exacta de las ventanas mostradas por el software. Todas estas discrepancias son ligeras y no influyen en que el lector no pueda seguir el tutorial.

Este tutorial no describe cómo utilizar el sistema operativo instalado en la computadora. Suponemos que el lector ya sabe realizar acciones tales como ejecutar programas, operar un ratón, mover, cambiar el tamaño, minimizar y maximizar ventanas, así como crear directorios (carpetas) y archivos, entre otras cosas. Un lector que desconozca estos procedimientos ha de aprender a usar el sistema operativo de la computadora antes de proseguir.

B.1.1 PRIMEROS PASOS

Cada circuito o subcircuito lógico que se diseña en Quartus II se llama *proyecto.* El software trabaja en un proyecto a la vez y guarda toda la información correspondiente en un solo directorio del sistema de archivos (usamos el término tradicional *directorio* para designar una ubicación en el sistema de archivos, pero Windows de Microsoft utiliza la palabra *carpeta*). Para comenzar el diseño de un circuito lógico nuevo, el primer paso consiste en crear un directorio para almacenar los archivos respectivos. Como parte de la instalación del software Quartus II, se colocan algunos proyectos de ejemplo en un directorio llamado *qdesigns.* A fin de guardar los archivos de diseño de este tutorial, emplearemos un directorio llamado *tutorial1*. La ubicación y el nombre del directorio no son importantes; por consiguiente, el lector puede usar cualquier directorio válido.

Inicie el software Quartus II. Debe ver una pantalla parecida a la que se exhibe en la figura B.1. Esa pantalla se compone de varias ventanas que dan acceso a todas las funciones del programa, las cuales el usuario selecciona con el ratón.

La mayor parte de los comandos que ofrece Quartus II puede accederse utilizando una serie de menúes que se hallan debajo de la barra de título. Por ejemplo, en la figura B.1 al hacer clic con el botón izquierdo del ratón en el menú llamado File (Archivo) se abre el menú mostrado

Figura B.1 Pantalla principal de Quartus II.

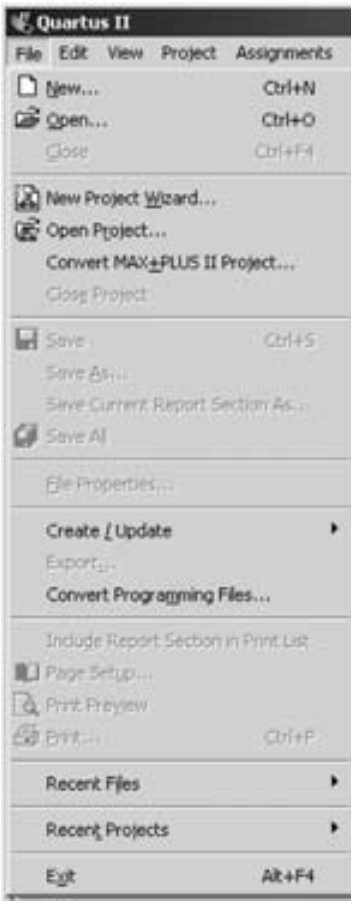


Figura B.2 Un ejemplo del menú File.

en la figura B.2. Si se hace clic con el botón izquierdo del ratón en la opción Exit, el programa se cierra. En general, siempre que el ratón se emplea para seleccionar algo, se utiliza el botón *izquierdo*. Por tanto, normalmente no especificaremos qué botón oprimir. En los pocos casos que sea necesario utilizar el botón *derecho*, lo señalaremos de manera explícita. Para algunos comandos es necesario acceder a dos o más menúes en secuencia. Usamos la convención Menú1 | Menú2 | Opción para indicar que a fin de seleccionar el comando deseado el usuario primero debe hacer clic con el botón izquierdo del ratón en el Menú1, luego dentro de este menú hacer clic en el Menú2 y después dentro del Menú2 hacer clic en Opción. Por ejemplo, File | Exit permite salir del programa Quartus II utilizando el ratón. Muchos comandos de Quartus II tienen un icono asociado que se muestra en alguna de las barras de herramientas. Para ver la lista de barras de herramientas seleccione Tools | Customize | Toolbars. Una vez que se abre una barra de herramientas, puede moverla con el ratón; los iconos de las barras pueden arrastrarse de una barra a otra. Para ver el comando de Quartus II asociado a un ícono, coloque el cursor del ratón sobre el ícono y aparecerá un recuadro que exhibe el nombre del comando.

Es posible modificar la apariencia de la pantalla de Quartus II que aparece en la figura B.1. En la sección B.6 mostramos cómo mover, cambiar el tamaño, cerrar y abrir ventanas dentro de la pantalla principal de Quartus II.

Ayuda en línea de Quartus II

Quartus II ofrece una documentación global en línea que responde a muchas de las preguntas que pueden surgir cuando se utiliza el software. La documentación está disponible en el menú de la ventana Help. Para formarse una idea de la extensión de la documentación provista vale la pena que el lector explore los temas de ayuda. Por ejemplo, si selecciona Help | How to Use Help hallará indicaciones de qué tipo de ayuda se brinda.

El usuario puede explorar rápidamente los temas de la ayuda seleccionando Help | Search, con lo que se abre un cuadro de diálogo donde pueden escribirse palabras clave. Otro método, la ayuda sensible al contexto, permite encontrar en poco tiempo documentación para temas específicos. Al trabajar con cualquier aplicación, si oprime la tecla de función F1 en el teclado se abre una pantalla de ayuda que muestra los comandos disponibles para esa aplicación.

B.2 CÓMO EMPEZAR UN PROYECTO NUEVO

Para empezar a trabajar en un diseño nuevo primero hay que definir un *proyecto de diseño.* Quartus II facilita la tarea al diseñador apoyándolo con un *asistente.* Seleccione File | New Project Wizard para llegar a una ventana que indica las funciones de este asistente. Haga clic en Next para abrir la ventana que se muestra en la figura B.3. Establezca *tutorial1\designstyle1* como directorio de trabajo. El proyecto debe tener un nombre, que puede ser el mismo que el del directorio. Hemos elegido el nombre *example_schematic* porque nuestro primer ejemplo implica la captura del diseño por medio de una captura esquemática. Obsérvese que Quartus II sugiere automáticamente que el nombre *example_schematic* sea también el nombre de la entidad de diseño de mayor nivel del proyecto. Ésta es una sugerencia razonable, pero puede ignorarse si el usuario desea emplear un nombre distinto. Haga clic en Next. Como aún no hemos creado el directorio *tutorial1\designstyle1*, Quartus II despliega el cuadro de mensaje de la figura B.4, que pregunta si quiere crear el directorio. Haga clic en Yes, lo cual conduce a la ventana de la figura B.5. En esta ventana el diseñador puede especificar qué archivos existentes (si los hay) deben incluirse en el proyecto. No tenemos archivos existentes, así que haga clic en Next.

Ahora aparece la ventana de la figura B.6, la cual permite al diseñador especificar las herramientas CAD de otras compañías (por ejemplo, las que no forman parte del software Quartus II) que deben utilizarse. En este libro hemos empleado el término *herramientas CAD* para referirnos a los paquetes de software desarrollados para usarlos en tareas de diseño asistidas por computadora. Otro término para el software de este tipo es *herramientas EDA,* donde el acrónimo significa *automatización del diseño electrónico.* Este término se usa en los mensajes de Quartus II que se refieren a herramientas de otras empresas, es decir, herramientas desarrolladas y comercializadas por compañías distintas a Altera. Como nos basaremos únicamente en Quartus II, no elegiremos ninguna otra herramienta.

Haga clic en Next para ir a la ventana mostrada en la figura B.7. Aquí podemos especificar el tipo de dispositivo en el que se implementará el circuito diseñado. Para el propósito de este

Figura B.3 Especificación del directorio y el nombre del proyecto.

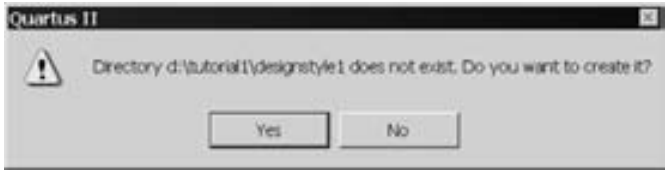


Figura B.4 Quartus II puede crear el directorio deseado.

tutorial la opción del dispositivo no tiene importancia. Elija la familia de dispositivos llamada Cyclone, que es un tipo de FPGA que usaremos en el apéndice C. No necesitamos elegir un dispositivo específico dentro de la familia Cyclone, así que haga clic en la opción No, I want to allow the Compiler to choose a device. Haga clic en Finish, con lo que volverá a la pantalla principal de Quartus II de la figura B.1, pero con *example_schematic* especificado como el nuevo proyecto.

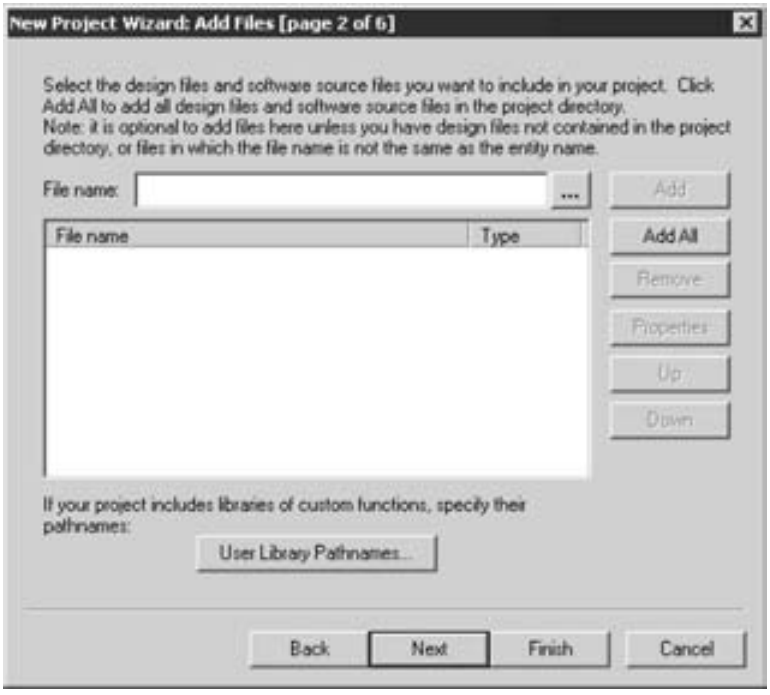


Figura B.5 Una ventana para incluir los archivos de diseño.

B.3 INGRESO DEL DISEÑO UTILIZANDO LA CAPTURA ESQUEMÁTICA

Como explicamos en el capítulo 2, los métodos de ingreso de diseño utilizados comúnmente incluyen la captura esquemática y el código de VHDL. En esta sección se ilustra cómo emplear la herramienta de captura esquemática provista en Quartus II, la cual se conoce como Editor de bloques (*Block Editor*). Como un ejemplo simple, dibujaremos un esquema para la función lógica $f = x_1x_2 + \bar{x}_2x_3$. En la figura 2.30 se mostró un diagrama de circuito para f y en la figura B.8*a* se reproduce. La tabla de verdad para f se proporciona en la figura B.8*b*. En el capítulo 2 también presentamos una simulación funcional. Después de crear el esquema, mostramos cómo usar el simulador de Quartus II a fin de comprobar la exactitud del circuito diseñado.

B.3.1 USO DEL EDITOR DE BLOQUES

El primer paso es trazar el esquema. En la pantalla principal de Quartus II seleccione File | New. Se abre una ventana, que se exhibe en la figura B.9, la cual permite al diseñador elegir el tipo de archivo que debe crearse. Los tipos de archivo posibles incluyen esquemas, código de VHDL y otros archivos de descripción de hardware como Verilog y AHDL (el HDL patentado por Altera). También es posible usar una herramienta de síntesis de otras empresas para generar un archivo

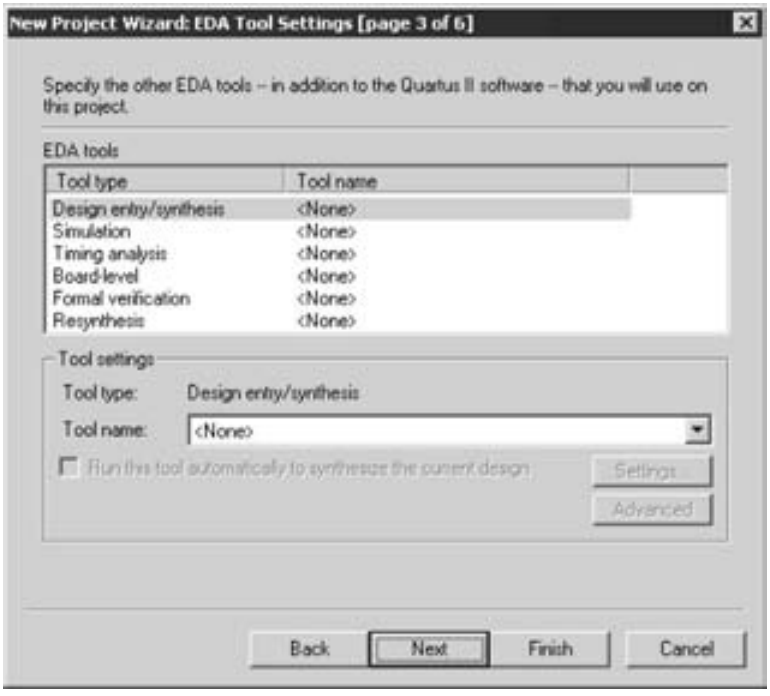


Figura B.6 Inclusión de otras herramientas EDA.

que represente el circuito en un formato estándar llamado EDIF (*Electronic Design Interface Format*, formato de interfaz de diseño electrónico). El estándar EDIF brinda un mecanismo práctico para que las herramientas EDA intercambien información. Como queremos ilustrar el método de ingreso esquemático en esta sección, elegimos la opción Block Diagram/Schematic File y hacemos clic en OK. Esta opción abre la ventana del editor de bloques mostrado en el lado derecho de la figura B.10. Al trazar un circuito en esta ventana se producirá el archivo de diagrama de bloques buscado.

Importación de los símbolos de las compuertas lógicas

El editor de bloques proporciona varias bibliotecas que contienen elementos del circuito, los cuales pueden importarse a un esquema. Para nuestro ejemplo simple, utilizaremos una biblioteca llamada *primitives,* que contiene compuertas lógicas básicas. Para acceder a la biblioteca haga doble clic en el espacio en blanco dentro de la pantalla del editor de bloques, de modo que se abra la ventana de la figura B.11 (otra forma de abrirla es seleccionando Edit | Insert Symbol o haciendo clic en el símbolo de la compuerta AND en la barra de herramientas). En esta figura, la lista desplazable etiquetada Libraries contiene varias bibliotecas incluidas en Quartus II. Para ampliar la lista haga clic en el pequeño símbolo + al lado de c:\quartus\libraries, luego haga clic en el símbolo + al lado de primitives y finalmente haga clic en el símbolo + al lado de logic. Ahora haga doble clic en el símbolo *and2* para importarlo al esquema (también puede hacer clic

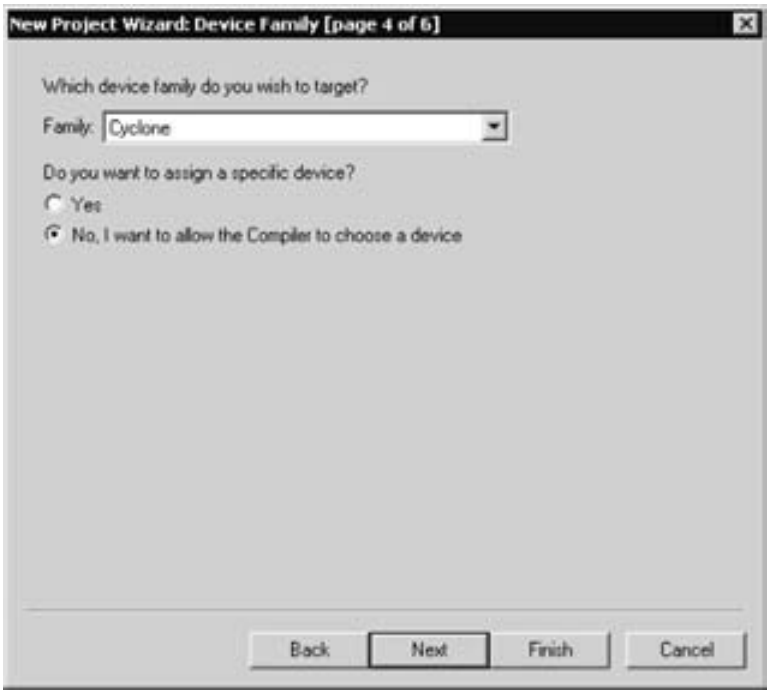
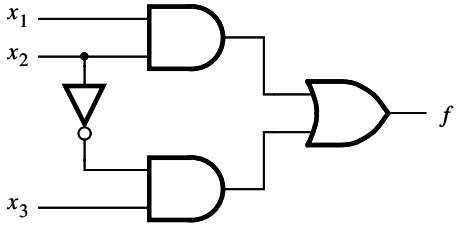


Figura B.7 Especificación de la familia de dispositivos.



a) Circuito

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

b) Tabla de verdad

Figura B.8 La función lógica de la figura 2.30.

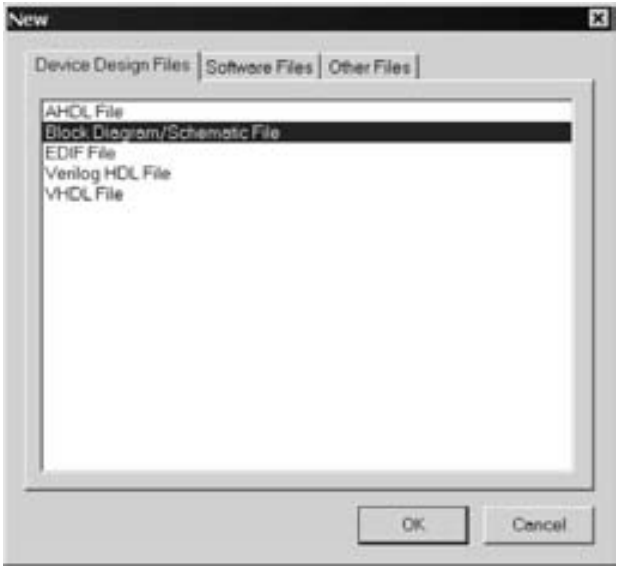


Figura B.9 Elección del tipo de archivo de diseño.

en *and2* y luego en OK). Un símbolo de compuerta AND de dos entradas aparece ahora en la ventana del editor de bloques. Con el ratón, mueva el símbolo a la posición donde debe aparecer en el diagrama y colóquelo ahí con un clic.

En un esquema cualquier símbolo puede seleccionarse utilizando el ratón. Coloque el puntero en la parte superior del símbolo de la compuerta AND en el esquema y haga clic para seleccionarlo. El símbolo se resalta en color. Para mover un símbolo, selecciónelo y, sin dejar de oprimir el botón del ratón, arrastre el ratón para mover el símbolo. Para facilitar la colocación de los símbolos gráficos puede mostrarse una cuadrícula de líneas guía en la ventana del editor de bloques seleccionando View | Show Guidelines.

La función lógica *f* requiere una segunda compuerta AND de dos entradas, una compuerta OR de dos entradas y una compuerta NOT. Siga los pasos descritos a continuación para importar estas compuertas al esquema.

Coloque el puntero del ratón sobre el símbolo de compuerta AND que ya se ha importado. Con la tecla Ctrl oprimida haga clic en el símbolo y arrástrelo. El editor de bloques importa de manera automática una segunda instancia del símbolo de compuerta AND. Este procedimiento es un atajo para duplicar un elemento de circuito cuando se precisen muchas instancias de él en el esquema. Desde luego, un método opcional es importar cada instancia del símbolo abriendo la biblioteca *primitives* como se describió antes.

Para importar el símbolo de compuerta OR, de nuevo haga doble clic en un espacio en blanco en el editor de bloques, de modo que se muestre la biblioteca *primitives*. Utilice la barra de desplazamiento para moverse hacia abajo por la lista de compuertas hasta encontrar el símbolo llamado *or2.* Impórtelo al esquema. Luego importe la compuerta NOT mediante el mismo procedimiento. Para orientar la compuerta NOT de modo que apunte hacia abajo, como se presenta en la figura B.8*a*, seleccione el símbolo correspondiente y luego utilice el comando Edit | Rotate

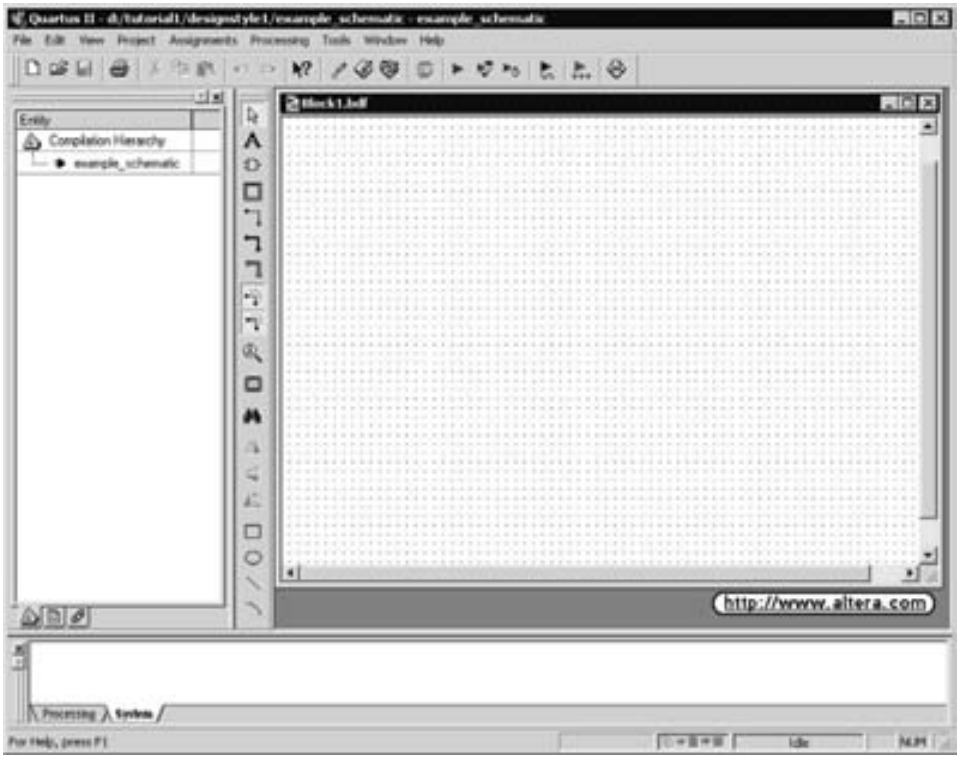


Figura B.10 Ventana del editor de bloques.

by Degrees | 270 para girar el símbolo 270 grados en sentido contrario al de las manecillas del reloj. Los símbolos del esquema pueden moverse seleccionándolos y arrastrando el ratón, como ya explicamos. Es posible seleccionar más de un símbolo al mismo tiempo haciendo clic y arrastrando el ratón hasta formar un contorno alrededor de los símbolos. Los símbolos seleccionados se mueven juntos al hacer clic en cualquiera de ellos y moverlo. Experimente con este procedimiento. Acomode los símbolos de modo que el esquema se vea similar al de la figura B.12.

Importación de los símbolos de entrada y salida

Ahora que hemos introducido los símbolos de las compuertas lógicas es necesario importar los que representan los puertos de entrada y salida del circuito. Abra la biblioteca *primitives* de nuevo. Desplácese por las compuertas hasta llegar a *pins.* Importe al esquema el símbolo llamado *input*. Luego importe dos instancias adicionales del símbolo de entrada. Para representar la salida del circuito abra la biblioteca *primitives* e importe el símbolo llamado *output.* Acomode los símbolos de manera que se dispongan de forma similar a la mostrada en la figura B.13.

Asignación de nombres a los símbolos de entrada y salida

Apunte a la palabra pin_name del símbolo de pin de entrada que se ubica en la esquina superior derecha del esquema y haga doble clic con el ratón. El nombre del pin queda seleccionado, lo que le permite escribir un nombre de pin nuevo. Teclee *x*1 como el nombre del pin. Oprima

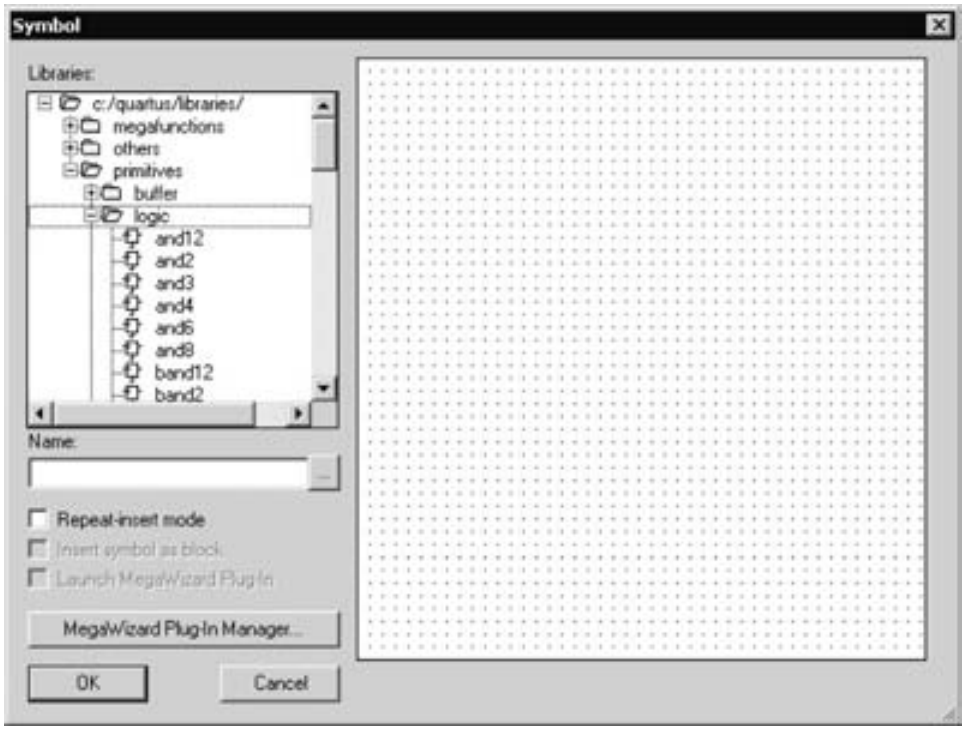


Figura B.11 Selección de los símbolos lógicos.

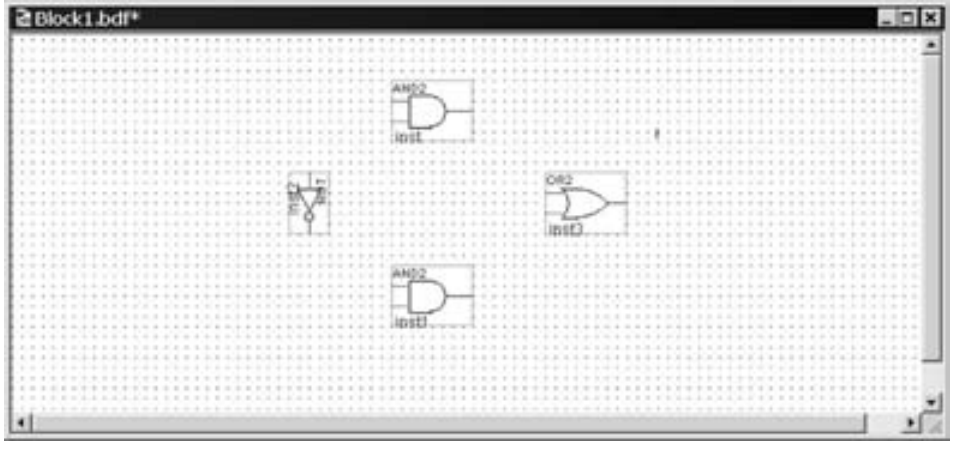


Figura B.12 Símbolos de compuerta importados.

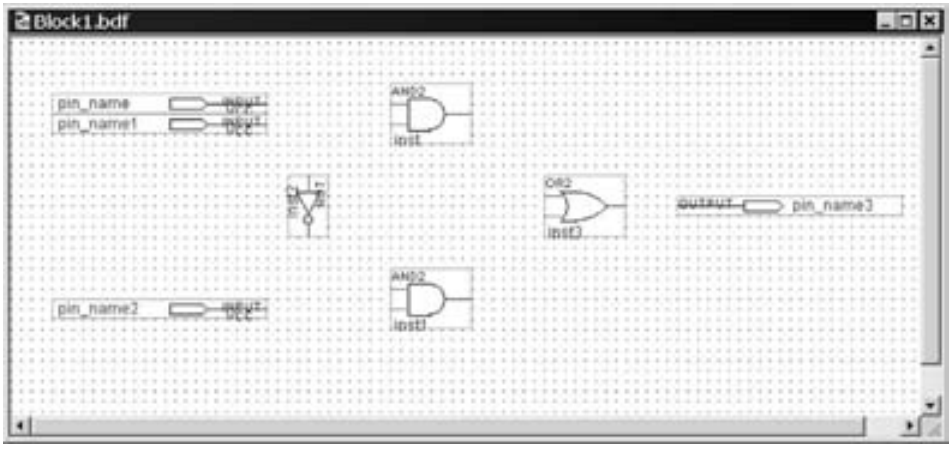


Figura B.13 El arreglo deseado de compuertas y pines.

la tecla Intro inmediatamente después de teclear el nombre del pin para que el ratón se mueva directamente debajo del pin al que está asignando el nombre. Este método puede usarse para nombrar cualquier pin. Asigne los nombres *x*2 y *x*3 a los pines de entrada medio e inferior, respectivamente. Por último asigne el nombre *f* al pin de salida.

Conexión de los nodos con los cables

El paso siguiente consiste en trazar líneas (cables) para conectar los símbolos en el esquema. Haga clic en el icono que parece una punta de flecha grande en la barra de herramientas vertical. Este ícono es la herramienta Selection and Smart Drawing, y permite al editor de bloques cambiar automáticamente entre los modos de seleccionar un símbolo en la pantalla o dibujar cables para interconectar símbolos. El modo apropiado se elige según el lugar a dónde el ratón apunta.

Coloque el puntero del ratón sobre el símbolo de entrada *x*1. Cuando apunta a cualquier parte del símbolo excepto al borde derecho, el puntero del ratón aparece como puntas de flecha en cruz. Esto indica que el símbolo se seleccionará si el botón del ratón se oprime. Mueva el ratón para apuntar a la pequeña línea, llamada *pinstub,* en el borde derecho del símbolo de entrada *x*1. El puntero del ratón cambia a una cruz, lo que permite dibujar un cable para conectar el pinstub con otro lugar en el esquema. Una conexión entre dos o más pinstubs en un esquema se llama *nodo.* El nombre deriva de la terminología eléctrica, donde la palabra *nodo* se refiere a cualquier número de puntos en un circuito que están conectados entre sí por medio de cables.

Conecte el símbolo de entrada para *x*1 a la compuerta AND en la parte superior del esquema como sigue. Mientras el ratón esté apuntando al pinstub del símbolo *x*1, haga clic y mantenga oprimido el botón del ratón. Arrastre éste a la derecha hasta que la línea (cable) que se está dibujando llegue al pinstub en la entrada superior de la compuerta AND; luego suelte el botón. Los dos pinstubs están ahora conectados y representan un solo nodo en el circuito.

Siga el mismo procedimiento para trazar un cable desde el pinstub del símbolo de entrada *x*2 a la otra entrada de la compuerta AND. Luego dibuje un cable desde el pinstub de la entrada de la compuerta NOT hacia arriba hasta que llegue al cable que conecta *x*2 a la compuerta AND.

Suelte el botón del ratón y observe que se traza de forma automática un punto de conexión. Los tres pinstubs correspondientes al símbolo de entrada *x*2, la entrada de la compuerta AND y la entrada de la compuerta NOT representan ahora un solo nodo en el circuito. La figura B.14 muestra una vista ampliada de la parte del esquema que contiene las conexiones trazadas hasta ahora. Para aumentar o disminuir la parte del esquema mostrado en la pantalla, utilice el ícono de lupa que se halla en la barra de herramientas.

Para completar el esquema, conecte la salida de la compuerta NOT a la compuerta AND inferior y el símbolo de entrada para *x*3 también al de la compuerta AND. Conecte las salidas de las dos compuertas AND a la compuerta OR y la compuerta OR al símbolo de salida *f*. Si comete algún error mientras conecta los símbolos puede seleccionar los cables donde está el error y luego borrarlos oprimiendo la tecla Supr o eligiendo la opción Edit | Delete. El esquema terminado se representa en la figura B.15. Guárdelo con File | Save As y elija el nombre *example_schematic.* Nótese que el archivo guardado se llama *example_schematic.bdf.*

Intente reordenar la disposición del circuito seleccionando y moviendo alguna compuerta. Obsérvese que a medida que mueve el símbolo de compuerta todos los cables de conexión se ajustan de manera automática. Esto ocurre porque Quartus II tiene una función llamada *movimiento de banda de goma (rubberbanding)* que se activó de forma predeterminada cuando usted eligió utilizar la herramienta *Selection and Smart Drawing*. Hay un ícono de movimiento de banda de goma, el ícono de la barra de herramientas que parece un cable en forma de L con pequeñas marcas de reloj en la esquina. Este ícono está resaltado para indicar el uso del movimiento de banda de goma. Inhabilítelo y mueva una de las compuertas para ver el efecto de esta función.

Como nuestro esquema de ejemplo es muy simple, resulta fácil trazar todos los cables del circuito sin producir un diagrama desordenado. Sin embargo, en los esquemas grandes algunos nodos que deben conectarse pueden estar muy apartados, caso en el que resulta poco práctico trazar cables entre ellos. En tal situación los nodos se conectan asignándoles etiquetas en vez de dibujar cables. Consulte la ayuda para obtener una descripción más detallada.

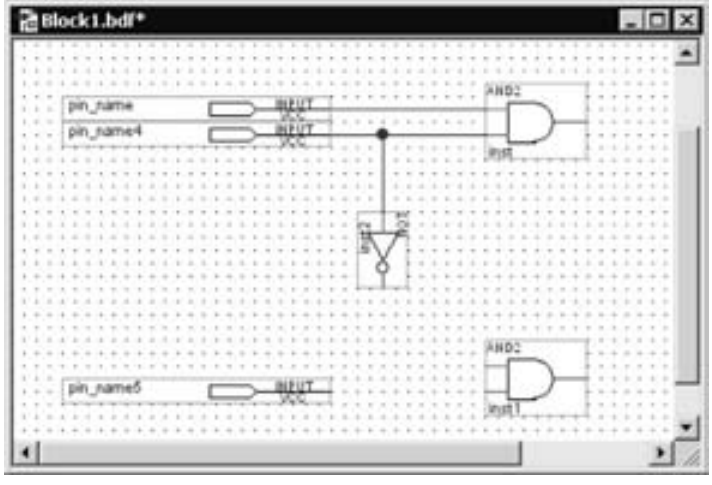


Figura B.14 Vista expandida del circuito.

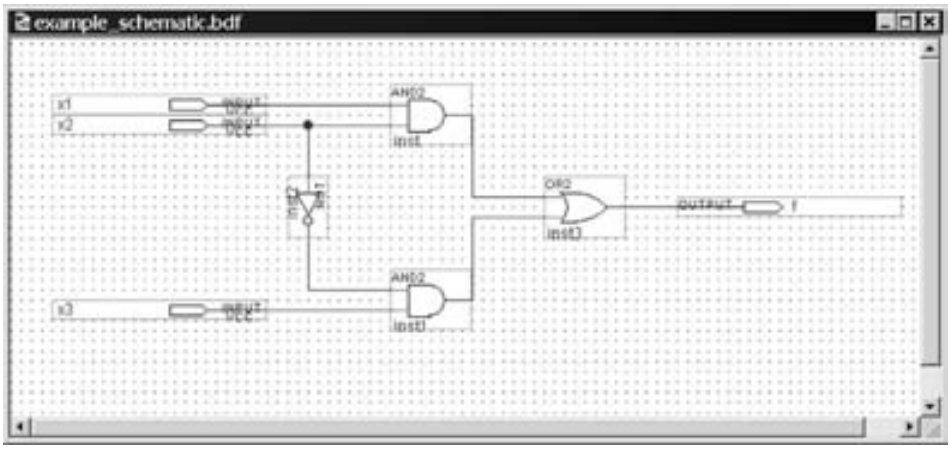


Figura B.15 El esquema terminado.

B.3.2 LA SÍNTESIS DE UN CIRCUITO A PARTIR DEL ESQUEMA

Luego de introducir un esquema en un sistema CAD, se le procesa por medio de varias herramientas CAD. En el capítulo 2 mostramos que el primer paso en el flujo CAD utiliza la herramienta de síntesis para traducir el esquema a expresiones lógicas. Por tanto, el paso siguiente en el proceso de síntesis, llamado *tecnología de mapeo,* determina cómo debe implementarse cada expresión lógica en los elementos lógicos disponibles en el chip objetivo.

Uso del compilador

Las herramientas CAD disponibles en Quartus II se dividen en varios módulos. Seleccione Tools | Compiler Tool para abrir la ventana de la figura B.16, la cual enumera cinco de los módulos principales. El módulo de *Analysis & Synthesis* realiza el paso de síntesis en Quartus II. Produce un circuito de elementos lógicos, donde cada uno de ellos puede implementarse directamente en el chip objetivo. El módulo *Fitter* (Instalador) determina la ubicación exacta en el

Figura B.16 La ventana de la herramienta Compiler.

chip donde se implementará cada uno de estos elementos producidos por la síntesis. Un análisis detallado de los módulos CAD se proporciona en el capítulo 12.

Estos módulos de Quartus II son controlados por un programa de aplicación llamado *Compiler*, que sirve para ejecutar un solo módulo a la vez o puede invocar múltiples módulos en secuencia. Hay varias formas de acceder al compilador en la interfaz de usuario de Quartus II. En la figura B.16, al hacer clic en el botón del extremo izquierdo del recuadro Analysis & Synthesis se ejecutará este módulo. De modo similar, el módulo *Fitter* puede ejecutarse haciendo clic en el botón del extremo izquierdo del recuadro. Al hacer clic en el botón Start se ejecutan los módulos de la figura B.16 en secuencia.

Otra forma práctica de tener acceso al *Compiler* es utilizando el menú Processing | Start. El comando para ejecutar el módulo de síntesis es Processing | Start | Start Analysis & Synthesis. Parte del módulo de síntesis también puede invocarse por medio del comando Processing | Start | Start Analysis & Elaboration. Este comando ejecuta sólo la primera parte de la síntesis, la cual revisa el proyecto de diseño en busca de errores de sintaxis e identifica los nombres de subdiseño más importantes que están presentes en el proyecto. El comando Processing | Start Compilation equivale a hacer clic en el botón Start de la figura B.16. También hay un ícono en la barra de herramientas para este comando, que se ve como un triángulo púrpura.

Una manera eficiente de usar las herramientas CAD es ejecutar sólo los módulos que se necesitan en cualquier fase concreta del proceso de diseño. Este método es pragmático, ya que algunas de las herramientas CAD pueden requerir muchas horas para concluir cuando se procesa un proyecto de diseño grande. Para el propósito de este tutorial, queremos realizar la simulación funcional de nuestro esquema. Como sólo se necesita la salida de la síntesis para realizar esta tarea, únicamente ejecutaremos el módulo de síntesis.

Seleccione Processing | Start | Start Analysis & Synthesis, utilice el ícono correspondiente en la barra de herramientas o el atajo Ctrl-k. Conforme avanza la compilación, su avance se indica en la esquina inferior derecha de la pantalla de Quartus II y también en la ventana *Status* en la parte izquierda; si esta ventana no está abierta, puede abrirla seleccionando la opción View | Utility Windows | Status. La compilación satisfactoria (o insatisfactoria) se indica en un cuadro de aparición instantánea. Responda haciendo clic en OK y revise el informe de compilación

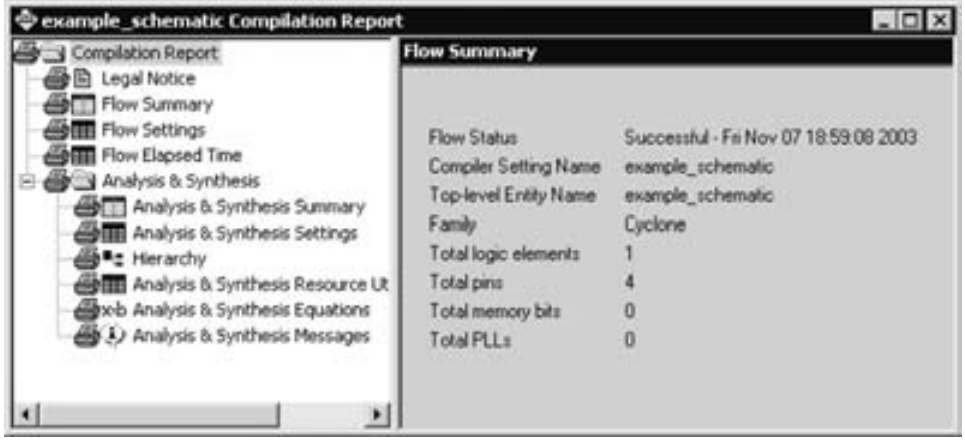


Figura B.17 Resumen del informe de compilación.

mostrado en la figura B.17 (si el informe aún no está abierto, puede abrirlo mediante un clic en el ícono Report en la ventana *Compiler Tool*, usando el ícono de la barra de herramientas correspondiente que parece una hoja en blanco en la parte superior de un chip azul o eligiendo Processing | Compilation Report). El resumen del informe muestra que nuestro pequeño diseño sólo empleará cuatro pines y un elemento lógico en un FPGA de Cyclone.

El informe de compilación ofrece mucha información que puede ser de interés para el diseñador. Por ejemplo, es posible ver la implementación detallada en forma de expresiones lógicas sintetizadas haciendo clic en el pequeño símbolo +, al lado de Analysis & Synthesis en el informe de compilación, y luego eligiendo Analysis & Synthesis Equations. La ecuación que Quartus II utilizó para implementar nuestro circuito es

$$f = x_1(x_3 + x_2) + \bar{x}_1 x_3 \bar{x}_2$$

El informe indica que AND es &, OR es # y NOT es!. Ésta no es la expresión más simple que uno esperaría:

$$f = x_1 x_2 + \bar{x}_2 x_3$$

Pero las dos expresiones representan la misma función y las herramientas CAD no siempre muestran la forma más simple de las ecuaciones en el informe de compilación. Éste puede abrirse en cualquier momento utilizando uno de los métodos descritos líneas arriba.

Errores

Quartus II muestra mensajes producidos durante la compilación en la ventana *Messages*. Esta ventana se halla en la parte inferior de la pantalla del programa de la figura B.1. Si el esquema se dibuja correctamente, uno de los mensajes informará que la compilación fue satisfactoria y que no hay errores o advertencias.

Para ver qué sucede si se comete un error, elimine el cable que conecta la entrada $x3$ con la compuerta AND inferior y compile el esquema modificado. La compilación no será satisfactoria y se mostrarán dos mensajes de error. El primero indica al diseñador que la compuerta AND afectada no encuentra una fuente; el segundo, que hay un error y una advertencia. En un circuito grande puede ser difícil encontrar el lugar de un error. Quartus II brinda ayuda por medio de la cual si el usuario hace doble clic en el mensaje de error, la ubicación correspondiente (y la compuerta AND en nuestro caso) se resaltarán. Vuelva a conectar el cable eliminado y recompile el circuito corregido.

B.3.3 SIMULACIÓN DEL CIRCUITO DISEÑADO

Quartus II incluye una herramienta de simulación que sirve para simular el comportamiento del circuito diseñado. Antes que el circuito pueda simularse es preciso crear las formas de onda deseadas, llamadas *vectores de prueba,* a fin de representar las señales de entrada. Utilizaremos el editor de formas de onda (*Waveform Editor*) de Quartus II para trazar vectores de prueba.

Uso del editor de formas de onda

Abra la ventana del editor de formas de onda seleccionando File | New, con lo que se abre la ventana de la figura B.9. Haga clic en la ficha Other Files para llegar a la ventana exhibida en la figura B.18. Seleccione Vector Waveform File y haga clic en OK.

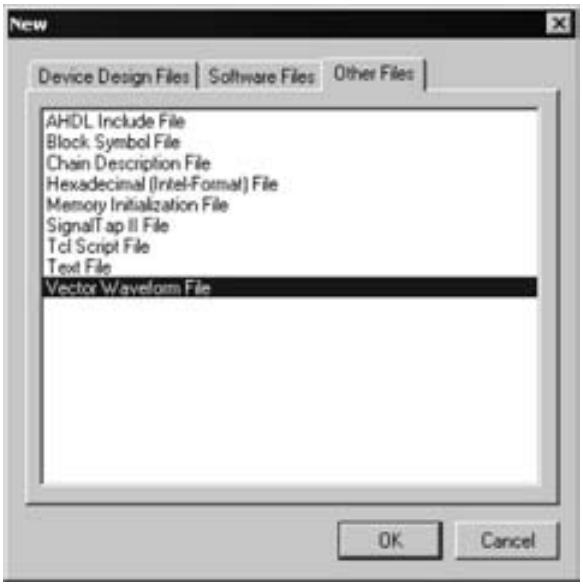


Figura B.18 Opción para preparar un archivo de vector de prueba.

La ventana del editor de formas de onda se presenta en la figura B.19. Guarde el archivo con el nombre de *example_schematic.vwf* y observe que el nombre de la barra de título de la ventana cambia. Indique que la simulación ha de ejecutarse de 0 a 160 ns eligiendo Edit | End Time e introduzca 160 ns en el cuadro de diálogo que se abre. Seleccione View | Fit in Window para mostrar todo el límite de la simulación, de 0 a 160 ns, en la ventana. Tal vez quiera aumentar al máximo el tamaño de la ventana.

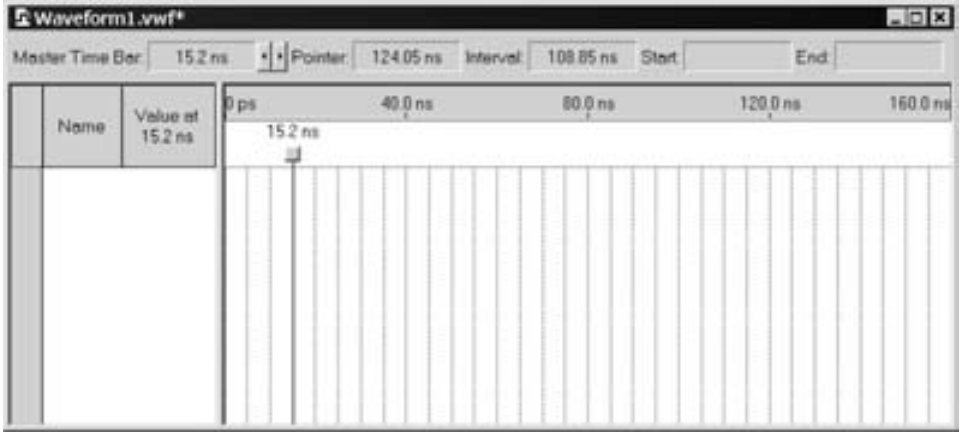


Figura B.19 La ventana del editor de formas de onda.

A continuación, queremos incluir los nodos de entrada y salida del circuito que se va a simular. Esto se realiza con la utilería *Node Finder*. Haga clic en Edit | Insert Node or Bus para abrir la ventana que aparece en la figura B.20. Puede escribir el nombre de una señal (pin) en el cuadro Name, pero es más conveniente hacer clic en el botón etiquetado Node Finder para abrir la ventana de la figura B.21. El *Node Finder* tiene un filtro utilizado para indicar qué tipo de nodos se van a buscar. Como estamos interesados en los pines de entrada y salida, establezca el filtro en Pins: all. Haga clic en el botón List para hallar los nodos de entrada y salida.

El *Node Finder* muestra en el lado izquierdo de la ventana los nodos *f*, *x*1, *x*2 y *x*3. Haga clic en *x*3 y luego en el signo > para añadirlo al cuadro *Selected Nodes* en el lado derecho de la figura. Repita el procedimiento para *x*2, *x*1 y *f*. Haga clic en OK para cerrar la ventana *Node Finder*, y luego en OK dentro de la ventana de la figura B.20. Esto deja una ventana del editor de formas de onda completamente desplegada, como se muestra en la figura B.22. Si usted no selecciona los nodos en el mismo orden que se muestra en esa figura, es posible volver a ordenarlos. Para mover una forma de onda hacia arriba o hacia abajo en la ventana del editor de formas de onda,

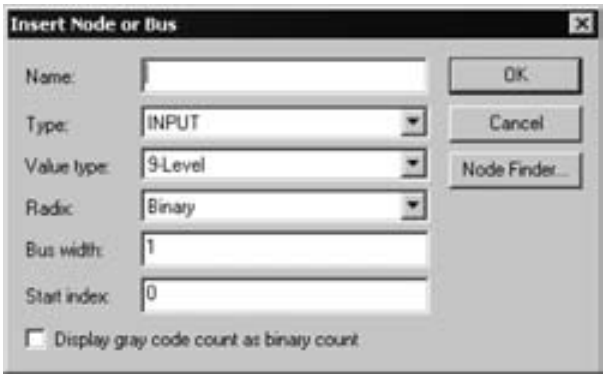


Figura B.20 Cuadro de diálogo de inserción de un nodo o bus (Insert Node or Bus).

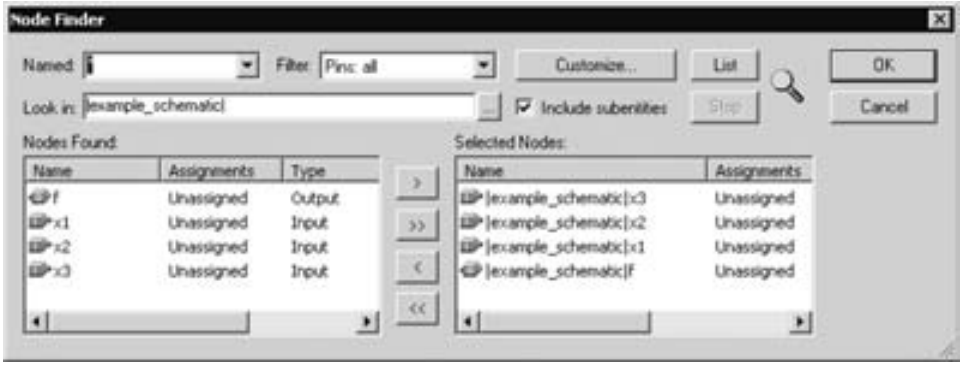


Figura B.21 Ventana del Node Finder.

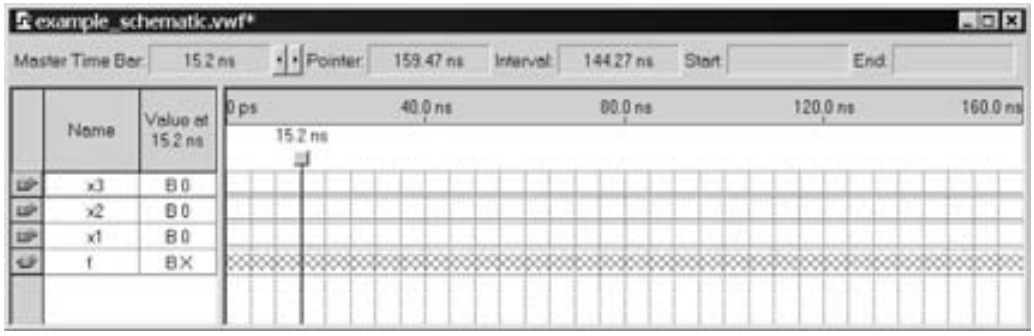


Figura B.22 Los nodos necesarios para la simulación.

haga clic en el nombre del nodo (en la columna *Name*) y suelte el botón del ratón. La forma de onda ahora está resaltada para mostrar la selección. Haga clic de nuevo en la forma de onda y arrástrela hacia abajo o hacia arriba en el editor de formas de onda.

Ahora especificaremos los valores lógicos que se van a utilizar para las señales de entrada durante la simulación. El simulador generará de manera automática los valores lógicos de la salida *f*. Para facilitar el dibujo de las formas de onda, Quartus II exhibe (en forma predeterminada) líneas guía verticales y brinda una función de dibujo que ajusta las ondas a esas líneas, las cuales pueden invocarse también eligiendo View | Snap to Grid. Nótese también una línea vertical continua, que puede moverse seleccionándola en la parte superior y arrastrándola de manera horizontal. Usaremos esta "línea de referencia" en el tutorial 2. Las formas de onda pueden trazarse con la herramienta *Selection*, que se activa seleccionando el ícono que parece una punta de flecha grande en la barra de herramientas vertical.

Para simular el comportamiento de un circuito grande es preciso aplicar un número suficiente de combinaciones de entrada y observar los valores esperados de las salidas. Puesto que el número de combinaciones de entrada posibles puede ser enorme, es necesario elegir una muestra relativamente pequeña (pero representativa). (El tema de la prueba del circuito se explora en el capítulo 11.) Nuestro circuito es muy pequeño, así que puede simularse por completo aplicando las ocho combinaciones posibles de las entradas *x*1, *x*2 y *x*3. Apliquemos una nueva combinación cada 20 ns. Para comenzar, todas las entradas son cero. En el punto 20 ns queremos que *x*3 pase a 1. Haga clic en *x*3; esto resalta la señal y activa la barra de herramientas vertical que permite moldear la forma de onda seleccionada. La barra de herramientas proporciona opciones tales como establecer la señal en 0, 1, incógnita (X), impedancia alta (Z), no-importa (DC) e invertir su valor existente (INV). Observe que la salida *f* se despliega como si tuviera un valor desconocido o incógnito en este momento, lo que se indica por medio de un patrón numerado. Un intervalo específico se selecciona haciendo clic sobre una forma de onda al principio del intervalo y arrastrándola al final de éste; el intervalo seleccionado se resalta. Elija el intervalo de 20 a 40 ns para *x*3 y establezca la señal en 1. Asimismo, establezca *x*3 en 1 de 60 a 80 ns, de 100 a 120 ns y de 140 a 160 ns. Enseguida establezca *x*2 en 1 de 40 a 80 ns y de 120 a 160 ns. Finalmente, establezca *x*1 en 1 de 80 a 160 ns. Complete estas asignaciones para obtener la imagen de la figura B.23 y guarde el archivo.

Un mecanismo práctico para cambiar las formas de onda de entrada es el que proporciona la herramienta *Waveform Editing*, cuyo ícono está en la barra de herramientas vertical y parece dos flechas que apuntan a la izquierda y a la derecha. Cuando el ratón se arrastra sobre algún intervalo de tiempo en el que la forma de onda es 0 (1), la forma de onda cambia a 1 (0). Experimente con esta función en la señal *x*3.

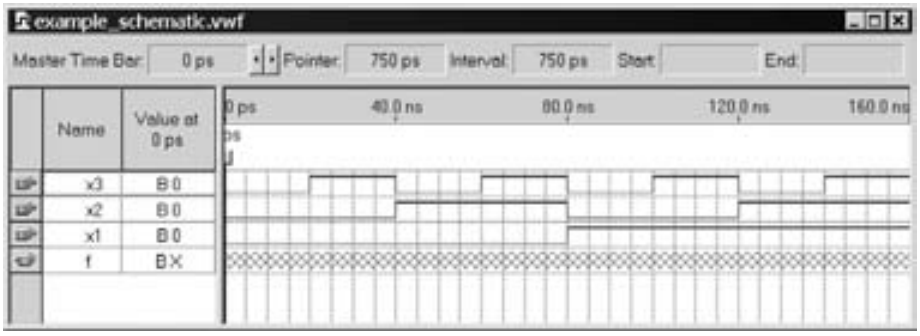


Figura B.23 Los vectores de prueba completos.

Ejecución de la simulación

Como explicamos en la sección 2.9.3, un circuito puede simularse de dos formas. La más sencilla es presuponer que los elementos lógicos y los cables de interconexión son perfectos, con lo cual no habrá ningún retraso en la propagación de las señales por el circuito. Esto se llama *simulación funcional.* Una alternativa más compleja es tomar en cuenta todos los retrasos de propagación, lo que conduce a la *simulación de tiempo.* En general, la simulación funcional se utiliza para comprobar la exactitud funcional —valga la expresión— de un circuito mientras se está diseñando. Esto requiere mucho menos tiempo, pues la simulación puede realizarse simplemente usando las expresiones lógicas que definen el circuito. En este tutorial sólo utilizaremos la simulación funcional. En el apéndice C abordamos la simulación de tiempo.

A fin de ejecutar la simulación funcional, seleccione Assignments | Settings para abrir la ventana *Settings*. En la parte izquierda de esta ventana haga clic en Simulator para desplegar la ventana de la figura B.24; seleccione Functional como el modo de simulación. Para completar la configuración del simulador elija el comando Processing | Generate Functional Simulation Netlist. El simulador de Quartus II toma las entradas de prueba y genera las salidas definidas en el archivo *example_schematic.vwf*. La ejecución de una simulación empieza seleccionando Processing | Start Simulation, o bien haciendo clic en el ícono de atajo de la barra de herramientas que parece un triángulo azul con una onda cuadrada debajo. Al final de la simulación, Quartus II indica la terminación satisfactoria y muestra un informe de simulación como el que aparece en la figura B.25. Como se advierte en la figura, el simulador crea una forma de onda para la salida *f*; toca al lector comprobar que corresponde a la tabla de verdad para la *f* dada en la figura B.8*b*.

Ahora hemos terminado nuestra introducción al diseño utilizando la captura esquemática. Seleccione File | Close Project para cerrar el proyecto en uso. A continuación mostraremos cómo usar Quartus II para implementar los circuitos especificados en VHDL.

B.4 INGRESO DEL DISEÑO CON VHDL

En esta sección se ilustra el proceso de utilizar Quartus II para implementar las funciones lógicas escribiendo código de VHDL. Implementaremos la función *f* de la sección B.3, donde

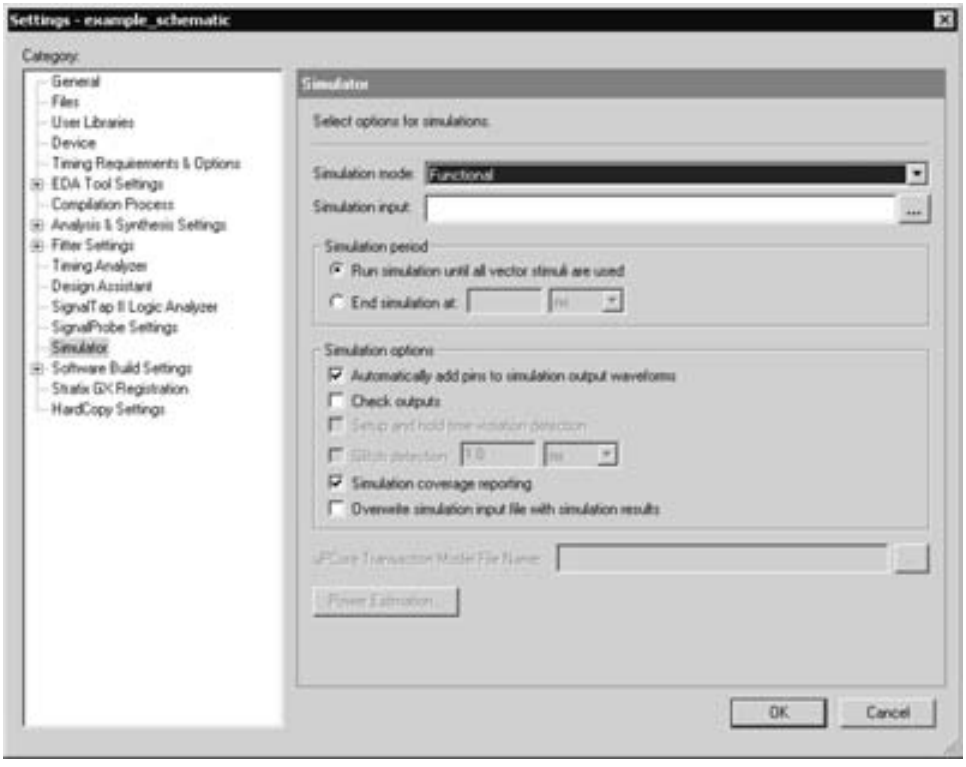


Figura B.24 Especificación del modo de simulación.

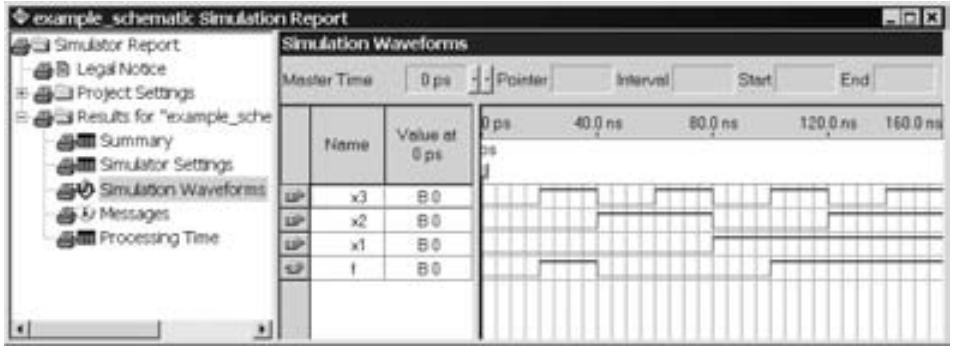


Figura B.25 El resultado de la simulación funcional.

empleamos la captura esquemática. Después de introducir el código de VHDL, lo simularemos funcionalmente.

B.4.1 CREACIÓN DE OTRO PROYECTO

Cree un proyecto nuevo para el diseño de VHDL en el directorio *tutorial1\designstyle2.* Utilice el asistente *New Project Wizard* para crear el proyecto según explicamos en la sección B.2. Llame al proyecto *example_vhdl* y elija la misma familia de chips FPGA para la implementación. Obsérvese que estamos creando este proyecto en un directorio nuevo, *designstyle2,* que es un subdirectorio del directorio *tutorial1.* Aun cuando podríamos haber creado un proyecto nuevo, *example_vhdl,* en el directorio anterior *designstyle1,* es recomendable crear proyectos distintos en directorios separados.

B.4.2 USO DEL EDITOR DE TEXTO

Quartus II ofrece un editor de texto que puede utilizarse para escribir código VHDL. Seleccione File | New para abrir la ventana de la figura B.9, elija la opción VHDL File y haga clic en OK. Esto abre la ventana del editor de texto (*Text Editor*). El primer paso consiste en especificar un nombre para el archivo que se creará. Elija File | Save As para abrir un cuadro de diálogo como el de la figura B.26. En el cuadro de lista desplegable Save as, haga clic en la opción VHDL File. En el cuadro etiquetado File name teclee *example_vhdl.* (Quartus II añadirá la extensión del nombre de archivo *vhd,* la cual debe emplearse para todos los archivos que contengan código de VHDL.) Deje activada la casilla Add file to current project en la parte inferior del cuadro de diálogo (y guarde el archivo). Esta opción informa a Quartus II que el archivo nuevo es parte del proyecto

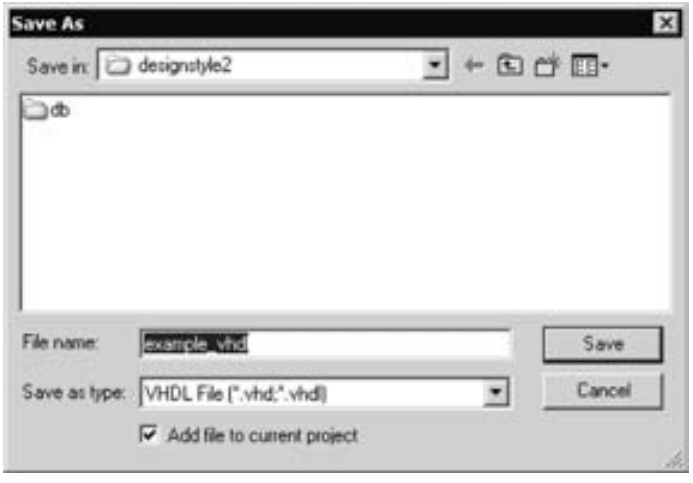


Figura B.26 Abriendo un archivo de VHDL.

actualmente abierto. Cabe aclarar que no es necesario usar el editor de texto incluido en Quartus II. Cualquier editor de texto sirve para crear el archivo llamado *example_vhdl.vhd*, siempre que pueda generar un archivo de texto plano (ASCII). Los archivos creados con algún editor de texto diferente al de Quartus pueden colocarse en el directorio *tutorial\designstyle2* e incluirse en el proyecto especificándolo así en la ventana del asistente *New Project Wizard* mostrada en la figura B.5 o identificándolo en la ventana Settings de la figura B.24, bajo la categoría Files.

El código de VHDL para este ejemplo se muestra en la figura 2.33. Introduzca este código en la ventana del editor de texto, con una pequeña modificación. En la figura 2.33 el nombre de la entidad es *example1*. Cuando creamos el proyecto nuevo elegimos el nombre *example_vhdl* para la entidad de diseño de alto nivel. Por consiguiente, la entidad en VHDL debe coincidir con este nombre. El código escrito debe aparecer como se muestra en la figura B.27. Guarde el archivo utilizando File | Save o el atajo Ctrl-s.

La mayor parte de los comandos del editor de texto se explica por sí sola. El texto se introduce en el *punto de inserción*, que se indica con una línea delgada vertical. El punto de inserción puede moverse utilizando las teclas de dirección del teclado o el ratón. Dos funciones del editor de texto son especialmente prácticas para escribir código de VHDL. Primera, el editor despliega diferentes tipos de instrucciones en diferentes colores, y segunda, el editor puede sangrar automáticamente el texto en una línea nueva que forma parte de la línea anterior. Estas opciones pueden controlarse por medio de la configuración del editor de texto en Tools | Options | Text Editor.

Uso de plantillas de VHDL

A veces la sintaxis del código de VHDL es difícil de recordar para un diseñador. Para ayudarle, el editor de texto ofrece una colección de plantillas de VHDL que brindan ejemplos de varios tipos de instrucciones de VHDL, como una declaración de entidad, una arquitectura y una instrucción de asignación de señal. Las plantillas también contienen algunos ejemplos de entidades completas de VHDL, como los contadores. Vale la pena examinarlos; hágalo seleccionando Edit | Insert Template | VHDL para conocer este recurso.

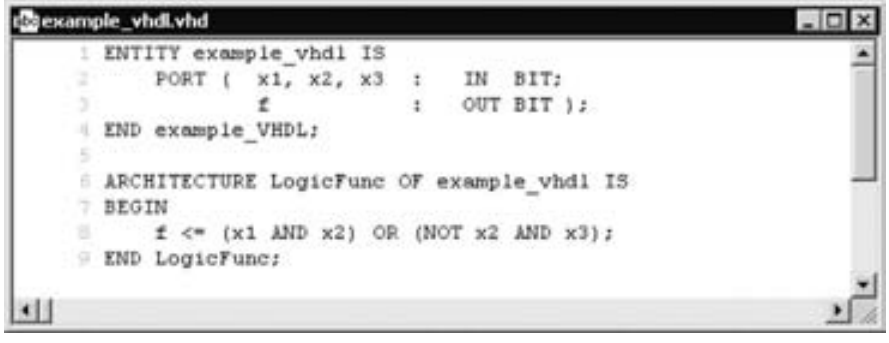


Figura B.27 El código de VHDL introducido en el editor de texto.

B.4.3 SÍNTESIS DE UN CIRCUITO A PARTIR DEL CÓDIGO DE VHDL

Igual que para un diseño creado a partir de la captura esquemática (sección B.3.2.), seleccione Processing | Start | Start Analysis and Synthesis (atajo: Ctrl-k) para que el compilador sintetice un circuito que implemente el código dado en VHDL. Si el código está bien escrito, el compilador muestra un mensaje que indica que no se generaron errores ni advertencias. El resumen del informe de compilación será, en esencia, como el mostrado en la figura B.17.

Si el compilador señala que hubo errores, por lo menos debe haber uno cometido al escribir el código de VHDL. En este caso, en la ventana Messages se exhibirán mensajes por cada error hallado. Si hace doble clic en un error, en la ventana del editor de texto se destacará la instrucción correspondiente en el código de VHDL. El compilador podría asimismo mostrar algunos mensajes de advertencia. Puede explorar los detalles como si se trataran de mensajes de error. Si selecciona un mensaje de advertencia o error y luego oprime la tecla F1 recibirá más información acerca de él.

B.4.4 EJECUCIÓN DE LA SIMULACIÓN FUNCIONAL

La simulación funcional del código de VHDL se hace exactamente igual que la simulación descrita páginas atrás para el diseño creado a partir de la captura esquemática. Cree un archivo nuevo del editor de formas de onda y elija File | Save As para guardar el archivo con el nombre *example_vhdl1.vwf.* Siguiendo el procedimiento descrito en la sección B.3.3, importe los nodos del proyecto al editor de formas de onda. Trace las formas de onda para las entradas $x1$, $x2$ y $x3$ que aparecen en la figura B.23. También es posible abrir el archivo de formas de ondas elaborado con anterioridad, *example_schematic.vwf,* y luego copiar y pegar las formas de onda para esas entradas. El método para copiarlas y pegarlas se describe en la ayuda del programa; en general, se trata del método normal de Windows para copiar y pegar. Cabe señalar que como el contenido de los dos archivos es idéntico, simplemente podemos hacer una copia de *example_schematic.vwf* y guardarla con el nombre *example_vhdl.vwf.*

Seleccione la opción *Functional Simulation* en la ventana semejante a la que se presenta en la figura B.24, y a continuación elija Processing | Generate Functional Simulation Netlist. Arranque la simulación. La forma de onda generada por el simulador, correspondiente a la salida *f,* debe ser igual que la mostrada en la figura B.25.

B.4.5 CÓMO USAR QUARTUS II PARA CORREGIR ERRORES EN EL CÓDIGO DE VHDL

En la sección B.3.2 indicamos que los mensajes mostrados sirven para ubicar y corregir de inmediato los errores de un esquema. Hay un procedimiento similar para hallar los errores en el código de VHDL. Para ilustrar esta función, abra el archivo *example__vhdl.vhd* con el editor de texto. En la octava línea, que es la instrucción de asignación de señal, borre el punto y coma que está al final. Luego guarde el archivo y ejecute el compilador con él. En la compilación se detectará ese error y se mostrarán los mensajes presentados en la figura B.28, los cuales indican que se identificó el problema al procesar la línea 9 del archivo fuente con el código de VHDL. Haga doble clic en este mensaje para situarse en la parte correspondiente del código. Se abrirá automáticamente la ventana del editor de texto con la línea 9 resaltada.

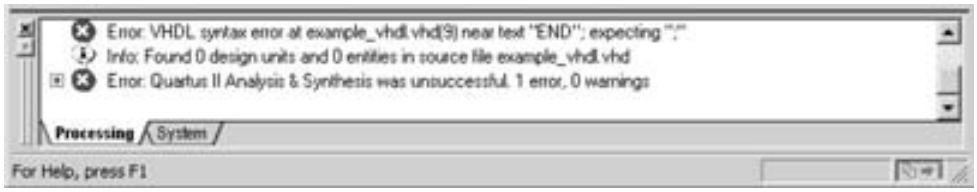


Figura B.28 La ventana de mensajes (*Message*).

Corrija el error insertando el punto y coma faltante; luego guarde el archivo y ejecute el compilador de nuevo para confirmar que ya no hay ningún error. Hemos terminado la introducción al diseño utilizando código de VHDL. Cierre este proyecto.

B.5 COMBINACIÓN DE MÉTODOS DE INGRESO DEL DISEÑO

Es posible diseñar un circuito lógico utilizando una combinación de métodos para la captura del diseño. Como ejemplo, diseñaremos un circuito que implementa la función

$$f = x_1x_2 + \bar{x}_2x_3$$

donde

$$x_1 = w_1w_2 + w_3w_4$$
$$x_3 = w_1w_3 + w_2w_4$$

Por consiguiente, el circuito tiene cinco entradas, x_2 y w_1 a w_4, y una salida f. Ya diseñamos un circuito para

$$f = x_1x_2 + \bar{x}_2x_3$$

en la sección B.3 siguiendo el método de ingreso esquemático. Para mostrar cómo pueden combinarse la captura esquemática y el código de VHDL, escribiremos código para las expresiones x_1 y x_3, y luego elaboraremos un esquema de alto nivel que conecte ese subcircuito de VHDL con el esquema creado en la sección B.3.

B.5.1 USO DE UN INGRESO ESQUEMÁTICO EN NIVEL ALTO

Siguiendo el método explicado en la sección B.2, cree un nuevo proyecto en un directorio llamado *tutorial1\designstyle3*. Use el nombre *example_mixed1* tanto para el proyecto como para la entidad de alto nivel. Para las pantallas del asistente *New Project Wizard* de las figuras B.5 a B.7, utilice la misma configuración usada en la sección B.2. Con el proyecto *example_mixed1* abierto, seleccione File | New para abrir la ventana de la figura B.9, y luego elija VHDL como el tipo de archivo por crear. Escriba el código de la figura B.29 y enseguida guarde el archivo con el nombre *vhdlfunctions.vhd*.

```
ENTITY vhdlfunctions IS
    PORT ( w1, w2, w3, w4  : IN   BIT ;
           g, h            : OUT  BIT ) ;
END vhdlfunctions ;

ARCHITECTURE LogicFunc OF vhdlfunctions IS
BEGIN
    g <= (w1 AND w2) OR (w3 AND w4);
    h <= (w1 AND w3) OR (w2 AND w4);
END LogicFunc ;
```

Figura B.29 Código de VHDL para el subcircuito *vhdlfunctions*.

A fin de incluir el subcircuito representado por *vhdlfunctions.vhd* en un esquema necesitamos crear un símbolo para este archivo que pueda importarse en el editor de bloques. Para hacerlo, seleccione File | Create/Update | Create Symbol Files for Current File. En respuesta, Quartus II genera un archivo de símbolos de bloque, *vhdlfunctions.bsf*, en el directorio *tutorial1\designstyle3*.

También queremos utilizar el circuito *example_schematic* creado en la sección B.2 como un subcircuito en el proyecto *example_mixed1*. De la misma manera que necesitamos hacer un símbolo para *vhdlfunctions*, se requiere un símbolo del editor de bloques para *example_schematic*. Seleccione File | Open y busque el archivo *tutorial1\designstyle1\example_schematic.bdf* para abrirlo. Ahora, elija File | Create/Update | Create Symbol Files for Current File. Quartus II generará el archivo *example_schematic.bsf* en el directorio *designstyle1*. Cierre el archivo *example_schematic.bdf*.

Ahora crearemos el esquema de alto nivel para nuestro proyecto de diseño combinado. Seleccione File | New y elija Block Diagram/Schematic File como el tipo de archivo que ha de crearse. Para guardar el archivo seleccione File | Save As y busque el directorio *tutorial1\designstyle3*. Es preciso buscar de nuevo nuestro directorio *designstyle3,* ya que Quartus II siempre recuerda el último directorio al que se ha tenido acceso; en el paso anterior creamos el archivo de símbolo *example_schematic.bsf* en el directorio *designstyle1*. Use el nombre *example_mixed1.bdf* cuando guarde el archivo de alto nivel.

Para importar los símbolos *vhdlfunctions* y *example_schematic*, haga doble clic en la pantalla del editor de bloques o seleccione Edit | Insert Symbol. Este comando abre la ventana que aparece en la figura B.30. Haga clic en el signo + al lado de la etiqueta Project en la esquina superior izquierda de la figura y luego haga clic en el elemento *vhdlfunctions* para seleccionarlo. Haga clic en OK para importar el símbolo al esquema. Ahora debemos importar el subcircuito *example_schematic*; como se almacena en el directorio de proyecto *designstyle1*, no aparece bajo la etiqueta Project en la figura B.30. Para encontrar el símbolo, desplácese por el cuadro Name: de la figura. Localice *example_schematic.bsf* en el directorio *tutorial1\designstyle1* y realice la operación de importación. Finalmente, importe los símbolos de entrada y salida de la biblioteca *primitives* y establezca las conexiones de los cables, como explicamos en la sección B.3, para obtener el circuito final representado en la figura B.31.

Compile el esquema. Si Quartus II produce un error e indica que no puede hallar el archivo de esquema *example_schematic.bdf*, entonces usted debe indicar a Quartus II dónde buscarlo. Seleccione Assignments | Settings para abrir la ventana *Settings*, la cual se mostró en la figura B.24.

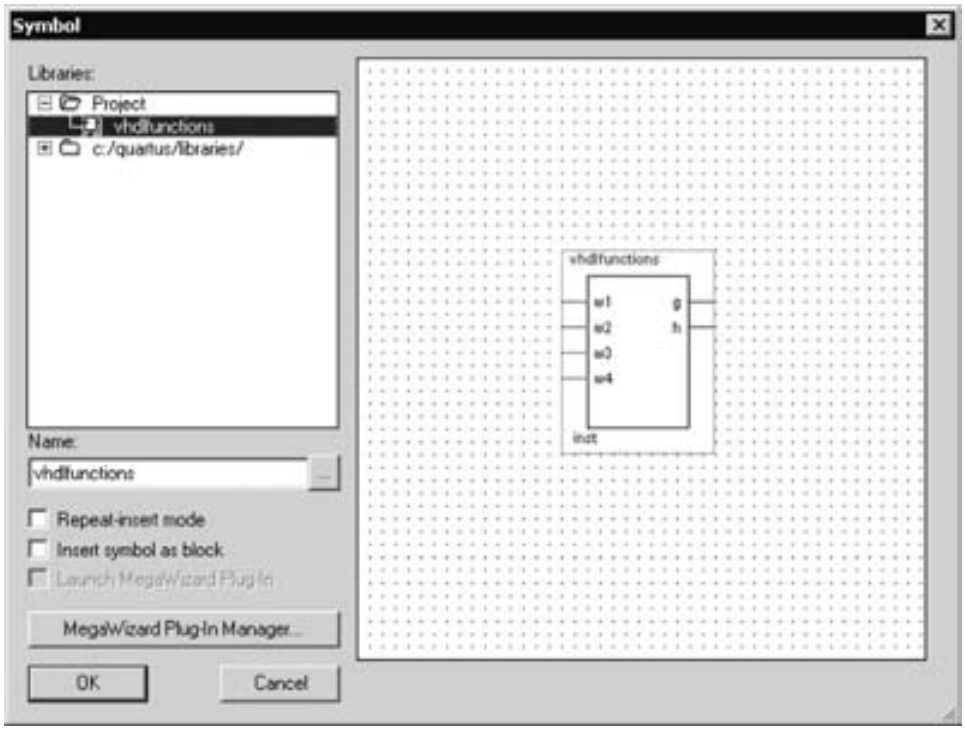


Figura B.30 Importación del símbolo para el subcircuito *vhdlfunctions*.

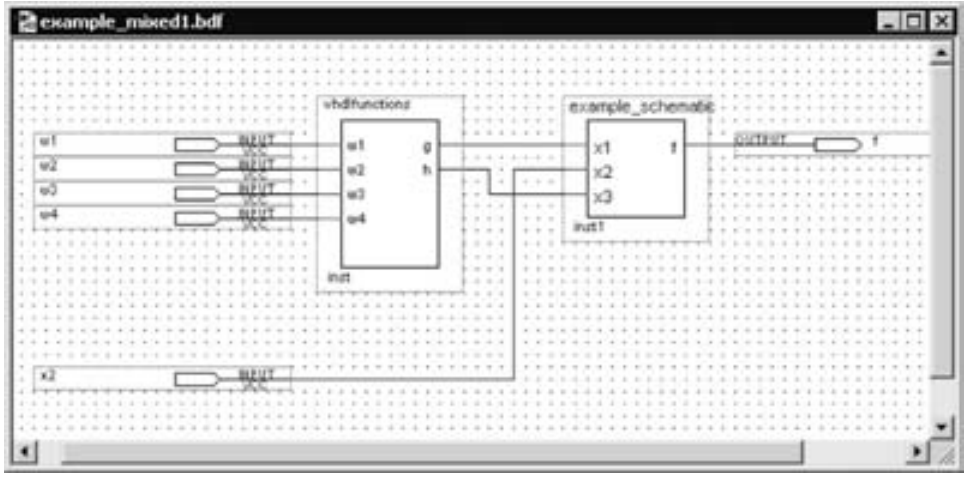


Figura B.31 El circuito completo.

En el lado izquierdo de esta ventana, haga clic en User Libraries y luego, en el cuadro de lista Library Name, busque el directorio *tutorial1\designstyle1*. Haga clic en Open para añadir este directorio al cuadro Libraries de la ventana *Settings*. Por último, haga clic en OK para cerrar la ventana *Settings* y luego intente compilar el proyecto de nuevo.

Para verificar su exactitud, el circuito debe simularse. Este circuito tiene cinco entradas, así que hay 32 combinaciones de entrada posibles que pueden probarse. En vez de ello, elegimos al azar sólo seis combinaciones, como se muestra en la figura B.32, y realizamos la simulación. Los valores correctos de *f*, los cuales son producidos por el simulador, se muestran en la figura. (En el capítulo 11 abordamos de manera pormenorizada los aspectos relativos a las pruebas y explicamos que utilizar un número relativamente pequeño de vectores de prueba en las entradas elegidos al azar es un método razonable.)

B.5.2 USO DE VHDL EN EL NIVEL ALTO

El ejemplo anterior muestra que un esquema puede incluir un símbolo que representa una entidad de VHDL. En la situación alternativa donde VHDL se utiliza para el archivo de diseño de alto nivel en un proyecto, el usuario tal vez quiera incluir un subcircuito que se ha diseñado previamente como un esquema. Una forma de hacerlo es utilizar software que traduzca el esquema en un archivo de VHDL. Quartus II incluye software como éste, al cual se accede desde el menú File. Para experimentar con esta función cierre el proyecto *example_mixed1* y abra el proyecto *example_schematic*, que se halla en el directorio *designstyle1*. Abra el archivo *example_schematic.bdf* y luego seleccione File | Create/Update | Create HDL Design for Current File. En la ventana que se abre, mostrada en la figura B.33, elija VHDL como el tipo de archivo fuente que ha de crearse y después haga clic en OK. Quartus II generará el archivo *example_schematic.vhd*. En la figura B.34 se muestra el contenido de este archivo (ligeramente editado para hacerlo más compacto). Nótese que Quartus II conservó los nombres originales de las entradas x_1, x_2 y x_3, y de la salida *f*. También eligió algunos nombres arbitrarios para los cables internos del circuito.

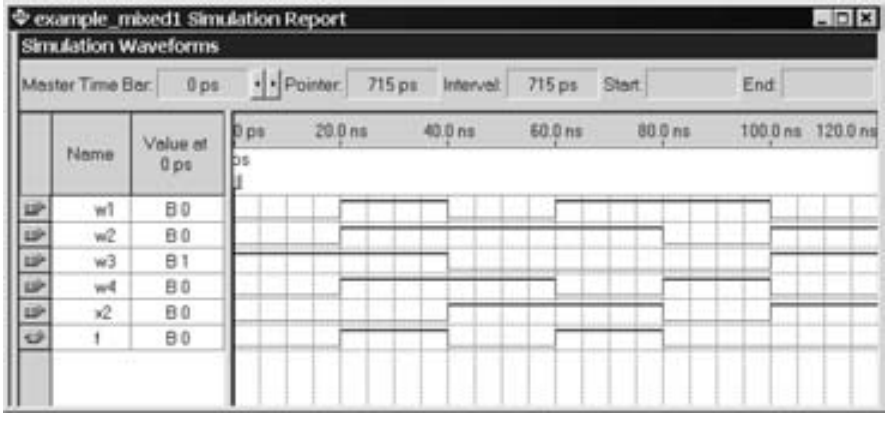


Figura B.32 Resultados de la simulación para el circuito *example_mixed1*.

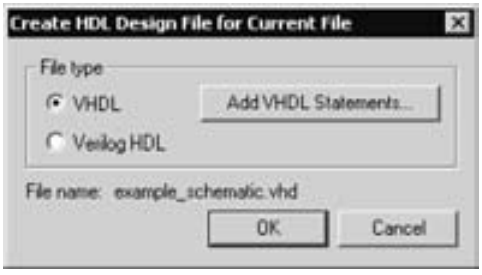


Figura B.33 Creando un archivo de VHDL para el esquema diseñado en la sección B.3.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY example_schematic IS
    PORT ( x1, x2, x3   : IN     STD_LOGIC;
           f            : OUT  STD_LOGIC );
END example_schematic;

ARCHITECTURE bdf_type OF example_schematic IS
    signal SYNTHESIZED_WIRE_0 : STD_LOGIC;
    signal SYNTHESIZED_WIRE_1 : STD_LOGIC;
    signal SYNTHESIZED_WIRE_2 : STD_LOGIC;
BEGIN
    SYNTHESIZED_WIRE_2 <= x1 AND x2;
    SYNTHESIZED_WIRE_1 <= SYNTHESIZED_WIRE_0 AND x3;
    f <= SYNTHESIZED_WIRE_1 OR SYNTHESIZED_WIRE_2;
    SYNTHESIZED_WIRE_0 <= NOT(x2);
END bdf_type;
```

Figura B.34 Código de VHDL para el circuito diseñado en la sección B.3.

El código de VHDL que escribimos en la sección B.4, presentado en la figura B.27, equivale al código generado automáticamente en la figura B.34. Puede instanciarse en una entidad de VHDL de alto nivel como se ilustra en la figura B.35. Esta entidad, llamada *example_mixed2,* implementa la misma función que diseñamos con la captura esquemática de la figura B.31. Mostramos cómo escribir este estilo de código de VHDL en los capítulos 4 y 5. El lector tal vez quiera crear un nuevo proyecto en Quartus II para este código, que luego puede compilar y simular utilizando los vectores de prueba de la figura B.32.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY example_mixed2 IS
    PORT ( w1, w2, w3, w4, x2  : IN     STD_LOGIC;
           f                   : OUT  STD_LOGIC );
END example_mixed2;

ARCHITECTURE Structure OF example_mixed2 IS
    COMPONENT example_schematic
        PORT ( x1, x2, x3      : IN     STD_LOGIC;
               f               : OUT  STD_LOGIC );
    END COMPONENT;
    COMPONENT vhdlfunctions
        PORT ( w1, w2, w3, w4  : IN     STD_LOGIC;
               g, h            : OUT  STD_LOGIC );
    END COMPONENT;
    SIGNAL g, h : STD_LOGIC;
BEGIN
    gandh: vhdlfunctions PORT MAP
        ( w1, w2, w3, w4, g, h );
    inst1: example_schematic PORT MAP
        ( g, x2, h, f );
END Structure;
```

Figura B.35 La entidad de VDHL de alto nivel para el ejemplo *example_mixed2*.

B.6 VENTANAS DE QUARTUS II

La pantalla de Quartus II contiene una serie de ventanas de programas auxiliares, las cuales pueden colocarse en varios lugares, cambiar de tamaño o cerrarse. En la figura B.36, se muestran cinco ventanas de Quartus II. La ventana del navegador de proyectos, *Project Navigator*, aparece cerca de la esquina superior izquierda de la figura. Bajo el encabezado *Compilation Hierarchy*, se representa una estructura tipo árbol del circuito diseñado que emplea los nombres de los módulos en el esquema de la figura B.31. Para ver la utilidad de esta ventana, abra el proyecto *example_mixed1* compilado previamente para abrir la pantalla correspondiente a la figura B.36. Ahora haga doble clic en el nombre *vhdlfunctions* en el navegador de proyectos. Quartus II abrirá automáticamente el archivo *vhdl_functions.vhd.* De igual modo, usted puede hacer doble clic en el nombre *example_schematic* y se abrirá el esquema correspondiente. La ventana *Status* se ubica debajo del navegador de proyectos. Como habrá observado, esta ventana muestra el progreso de la compilación mientras Quartus II compila un proyecto. En la parte inferior de la

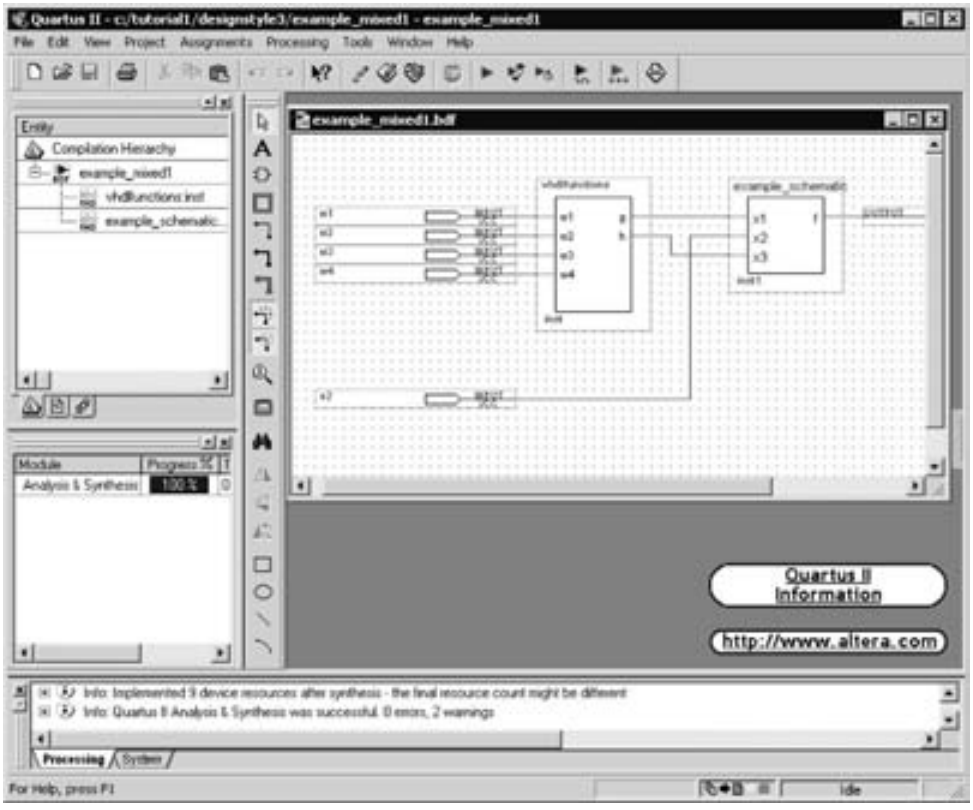


Figura B.36 La pantalla principal de Quartus II.

figura B.36 está la ventana de mensajes (*Message*), que muestra al usuario los mensajes producidos durante la compilación.

El área grande del lado derecho de la pantalla de Quartus II sirve a varios fines. Como hemos visto, la utiliza el editor de bloques, el editor de texto y el editor de formas de onda. También se ocupa para mostrar varios resultados de la compilación y de la simulación.

Una ventana de programas auxiliares puede moverse arrastrando su barra de título, y aumentarse o disminuirse arrastrando su borde o cerrase haciendo clic en la X que se halla en la esquina superior derecha. Una ventana de programas auxiliares se abre con la opción View | Utility Windows.

Los comandos disponibles en Quartus II son *sensibles al contexto,* según la herramienta del programa que se esté empleando. Por ejemplo, cuando el editor de texto se está usando, el menú *Edit* contiene un conjunto de comandos diferente del conjunto disponible cuando otra herramienta, digamos el editor de formas de onda, está en uso.

B.7 COMENTARIOS FINALES

En este tutorial hemos presentado el uso básico del sistema CAD Quartus II. Mostramos cómo realizar la captura del diseño al dibujar un esquema o escribir código de VHDL. También hemos ilustrado cómo pueden combinarse estos métodos de ingreso del diseño en un diseño jerárquico. Cada diseño se compiló y se simuló utilizando la simulación funcional.

En el tutorial siguiente describiremos módulos adicionales de Quartus II que sirven para implementar circuitos en PLD.

apéndice

C

TUTORIAL 2
IMPLEMENTACIÓN DE CIRCUITOS EN DISPOSITIVOS DE ALTERA

En este tutorial describimos cómo utilizar las herramientas de diseño físico de Quartus II. Además de los módulos usados en el tutorial 1, se presentan los siguientes módulos de Quartus II: *Fitter*, *Floorplan Editor* y *Timing Analyzer*. Para ilustrar los procedimientos descritos, primero implementaremos el proyecto *example_vhdl* creado en el tutorial 1 en un CPLD MAX 7000.

C.1 IMPLEMENTACIÓN DE UN CIRCUITO EN UN CPLD MAX 7000

Seleccione File | Open Project y examine el directorio *designstyle2*, que contiene el ejemplo de diseño de VHDL usado en el tutorial 1. Como se muestra en la figura C.1, elija el proyecto *example_vhdl* (los archivos de proyecto de Quartus II tienen la extensión de archivo *.qpf*) y haga clic en Open.

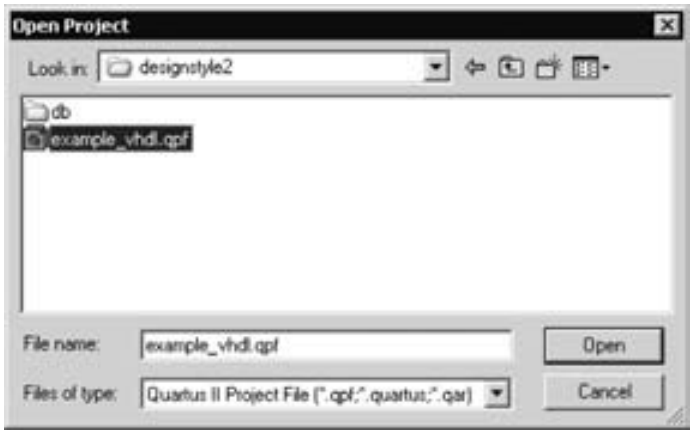


Figura C.1 Cuadro de diálogo para abrir el proyecto *example_vhdl*.

C.1.1 SELECCIÓN DE UN CHIP

En el tutorial 1 empleamos el compilador para realizar las operaciones de síntesis, las cuales generaron la información requerida para la simulación funcional. Ahora implementaremos el diseño en un CPLD y luego utilizaremos la simulación de tiempo.

Para especificar qué chip desea usar, seleccione Assignments | Device para abrir la ventana mostrada en la figura C.2. Para seleccionar la familia de dispositivos CPLD MAX 7000, haga clic en el menú desplegable dentro del cuadro etiquetado Family y elija MAX7000S. Esta S se refiere a los miembros de la familia MAX 7000 que son programables dentro del sistema. Los métodos de la programación en CPLD se estudian en el capítulo 3, sección 3.6.4. Obsérvese que en algunos casos Quartus II mostrará el mensaje "Device family selection has changed. Do you want to remove all pin assignments?". Haga clic en Yes para cerrar esta lista desplegable.

En el cuadro Target device puede especificar que Quartus II seleccione automáticamente un dispositivo durante la compilación. La posibilidad de tener un chip elegido en forma automática a veces es conveniente para el diseñador. Sin embargo, en este caso deseamos escoger un chip específico, así que haga clic en Specific device selected in 'Available devices' list.

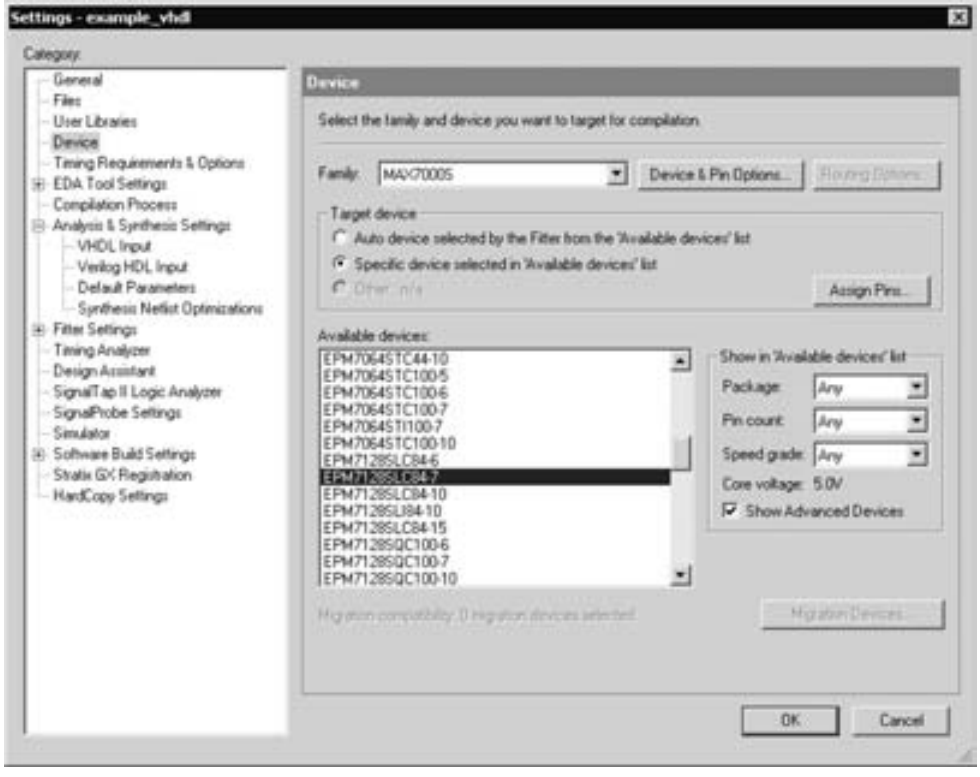


Figura C.2 Selección de un dispositivo de la familia MAX7000S.

Los chips disponibles en la familia MAX 7000S se muestran en el cuadro etiquetado Available devices. Un chip disponible es el EPM7128SLC84-7 [si este dispositivo no aparece en la lista, cambie el elemento Speed grade en el cuadro Show in 'Available devices' list a Any (cualquiera)]. El significado del nombre del chip es el siguiente: EPM7 significa que el chip es un miembro de la familia MAX 7000 y el 128 proporciona el número de macroceldas que tiene. El indicador LC84 significa un paquete PLCC de 84 pines; este tipo de paquete se describe en la sección 3.6.3. El −7 proporciona el *grado de velocidad* (*speed grade*). En el apéndice E se estudian los grados de velocidad. Como se indicó en la figura C.2, haga clic en el dispositivo EPM7128SLC84-7 y luego haga clic en OK para cerrar la ventana Settings. Hemos elegido este chip porque se incluye en una tarjeta de desarrollo de Altera que se estudia en el apéndice D.

C.1.2 COMPILACIÓN DEL PROYECTO

En el apéndice B simplemente ejecutamos las herramientas de síntesis de Quartus II con el comando Processing | Start | Start Analysis & Synthesis. Ahora deseamos ejecutar no sólo las herramientas de síntesis, sino también otras que implementan el circuito en el dispositivo objetivo. Para invocar todas las herramientas necesarias, seleccione Processing | Start Compilation, o utilice el icono de barra de herramientas que parece un triángulo morado sólido. Esta acción ejecuta en secuencia cuatro de los módulos mostrados en la figura B.16: *Synthesis*, *Fitter*, *Assembler* y *Timing Analyzer*. Como vimos en el tutorial 1, el avance de compilación a través de cada módulo de Quartus II aparece en la ventana de estado en el lado izquierdo de la pantalla del programa. Después de que el módulo *Analysis & Synthesis* convierte el código de VHDL en un circuito que se compone de macroceldas, el módulo *Fitter* les elige lugares en el dispositivo. En el capítulo 12 se presenta un análisis detallado de los módulos CAD.

Cuando la compilación termina, se produce el informe de compilación mostrado en la figura C.3. Como dijimos en el tutorial 1, hay mucha información útil ahí. Haga clic en el pequeño símbolo + para ampliar la sección *Fitter* del informe, y luego haga clic en la sección *Fitter Equations* para llegar a la pantalla de la figura C.4. Desplácese hasta esta parte del informe para ver

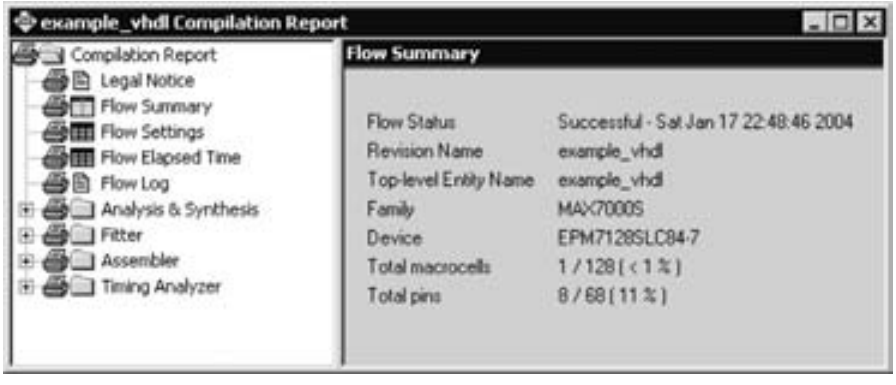


Figura C.3 El resumen de la compilación.

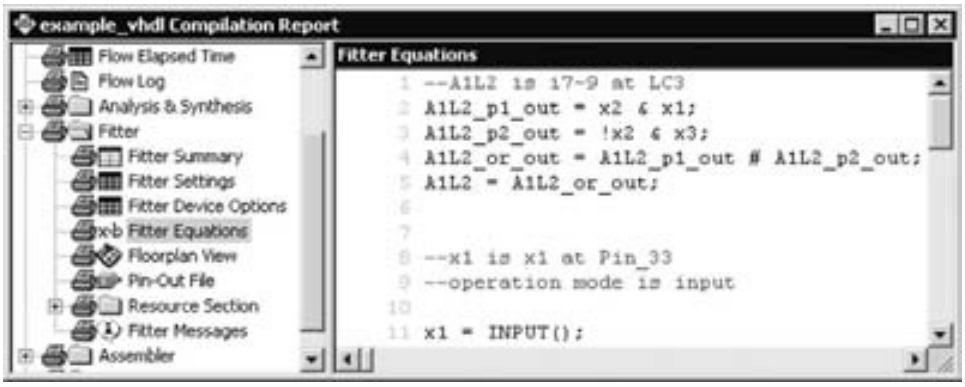


Figura C.4 La sección de ecuaciones del *Fitter* (*Fitter Equations*).

las expresiones lógicas implementadas por nuestro circuito. En la parte inferior del informe la salida *f* se da como

f = OUTPUT(A1L2);

Esto significa que *f* aparece en un pin de salida y que la salida se define por medio de la expresión lógica llamada A1L2, la cual se produce como se indica cerca de la parte superior de la sección *Fitter Equations* en la figura C.4. Estas expresiones implementan de manera adecuada nuestra función lógica $f = x_1x_2 + \bar{x}_2x_3$.

C.1.3 REALIZACIÓN DE LA SIMULACIÓN DE TIEMPO

La simulación de tiempo se realiza siguiendo el mismo procedimiento descrito en el tutorial 1 para la simulación funcional. Seleccione Assignments | Settings y haga clic en el elemento Simulator, como se muestra en la figura B.24. Abra la lista desplegable al lado de Mode Simulation y cambie este parámetro de Functional a Timing.

Utilice las formas de onda de entrada para $x1$, $x2$ y $x3$ trazadas con el editor de formas de onda en el tutorial 1 como entradas para la simulación de tiempo. Seleccione Processing | Start Simulation para ejecutar la simulación. Cuando ésta termina, aparece el informe de simulación. Parte de este informe se muestra en la figura C.5. Elija View | Fit in Window para ver el intervalo de tiempo completo de las formas de onda. Compare estas formas con las mostradas en la figura B.25. La simulación de tiempo produce los mismos resultados que la simulación funcional del tutorial 1, excepto porque las veces en que los cambios en *f* ocurren ahora están determinados por las características de la sincronización del chip EPM7128SLCS4-7.

Podemos usar la línea de referencia vertical en la pantalla para determinar el tiempo exacto en que *f* cambia de valor. Para hacerlo seleccione View | Snap to Transition, de modo que el puntero del ratón se alinee perfectamente con un borde en cualquier forma de onda. Haga clic y arrastre la línea de referencia vertical hasta el punto donde *f* primero cambia a 1, como se muestra en la figura. El cuadro etiquetado Master Time Bar ahora exhibe 27.5 ns, lo que significa que se requieren 7.5 ns para que el cambio en $x3$, el cual ocurre a los 20 ns, ocasione un cambio en *f*. Esto da como resultado una reflexión del grado de velocidad -7 del chip, que se especifica como un retraso de 7.5 ns desde un pin de entrada hasta un pin de salida.

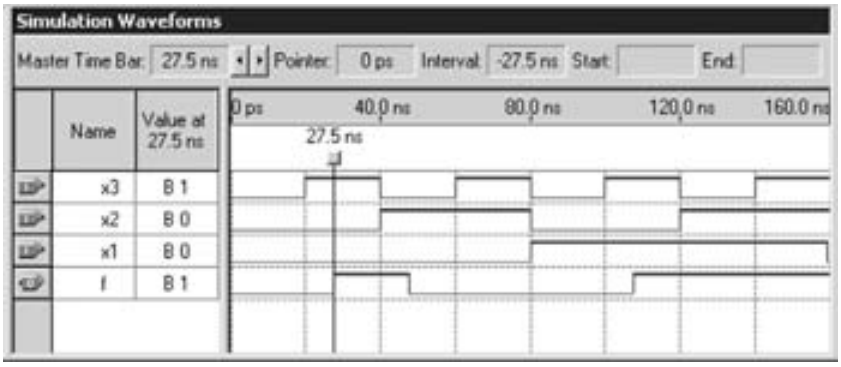


Figura C.5 El informe de simulación de tiempo.

C.1.4 USO DEL EDITOR DE PINES (FLOORPLAN EDITOR)

Además de examinar las ecuaciones en el informe de compilación, otra forma de ver los resultados de la implementación es utilizar el editor de pines (*Floorplan Editor*). Seleccione Assignments | Timing Closure Floorplan para abrir la ventana mostrada en la figura C.6. Otra manera de abrir esta ventana es haciendo clic en el icono correspondiente en la barra de herramientas. Para hacer que la ventana se parezca a la de la figura, tal vez sea necesario cambiar la configuración de la herramienta *Floorplan* eligiendo View | Interior Cells, lo que hace que se muestren las macroceldas del dispositivo. En la figura C.6 aparecen algunas de las macroceldas del chip EPM7128SLC84-7. Como se indica en el apéndice E, las macroceldas están organizadas en bloques de arreglos lógicos (LAB), cada uno de los cuales contiene 16 macroceldas. Para mostrar vistas más grandes o más pequeñas de los LAB, haga clic en los botones para aumentar el tamaño en la barra de herramientas vertical: de izquierda a derecha para agrandar la imagen y de derecha a izquierda para reducirla. Para mostrar diferentes secciones del chip utilice las barras de desplazamiento de la ventana.

El editor de pines utiliza diferentes colores para indicar las macroceldas usadas y sin usar en un circuito. Para nuestro pequeño ejemplo hay pines que se utilizan para las tres entradas al circuito, y una macrocelda proporciona la salida del circuito. Ajuste la pantalla de modo que la macrocelda que produce la salida *f* esté visible, según se presenta en la figura C.7. Haga clic en esta macrocelda para seleccionarla. Si elige View | Routing | Show Node Fan-In el editor de pines puede trazar líneas que indican a qué otras macroceldas está conectada la macrocelda. También es posible ver qué función lógica se implementó en el nodo seleccionado habilitando View | Equations. Como se advierte en la figura, esta opción muestra las expresiones lógicas del informe de compilación en la parte inferior de la ventana del editor de pines.

En vez de mostrar las macroceldas, la herramienta *Floorplan* puede desplegar de manera opcional una imagen de los pines del encapsulado. Para cambiar a esta vista, seleccione View | Package Top. Esto conduce a la pantalla mostrada en la figura C.8. Para cerrar el visor del archivo de informe de ecuaciones, seleccione de nuevo View | Equations para inhabilitar esta función.

La herramienta *Floorplan* no es esencial en el flujo CAD descrito con anterioridad. Sólo ofrece una vista gráfica de la información contenida en el informe de compilación. Describiremos un uso distinto de esa herramienta en el apéndice D, donde se utilizará para modificar los resultados de la implementación producidos por el programa de compilación en vez de simplemente mostrarlos.

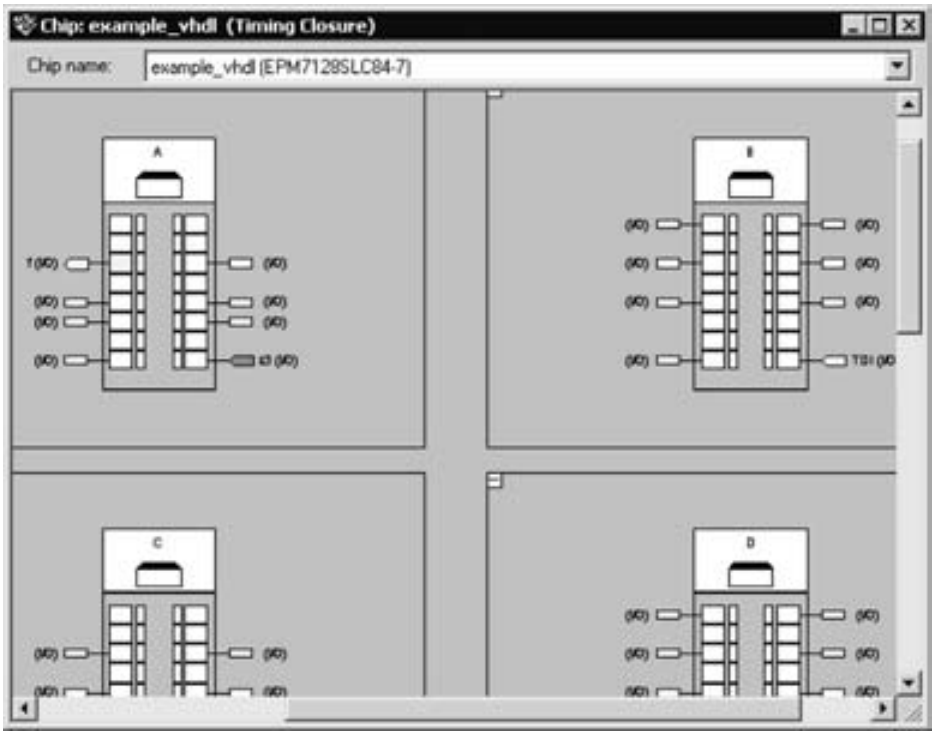


Figura C.6 La pantalla de finalización de tiempo del editor de pines (*Timing Closure Floorplan*).

Ahora hemos completado la implementación del proyecto *example_vhdl* en un chip MAX 7000. Cierre el proyecto.

C.2 IMPLEMENTACIÓN DE UN CIRCUITO EN UN FPGA CYCLONE

El flujo CAD utilizado para implementar un circuito en un FPGA Cyclone es el mismo que el empleado para el CPLD MAX 7000. En el capítulo 4 mostramos que la síntesis lógica multinivel es una estrategia de optimación eficaz cuando nos centramos en los diseños para los FPGA basados en LUT (*lookup tables*, tablas de consulta). En la figura 4.48 se proporciona el código de VHDL para una función lógica de siete variables usada para ilustrar los beneficios de la síntesis multinivel. En esta sección crearemos un proyecto de diseño nuevo, llamado *example_vhdl2,* que representa el código de VHDL de esa figura.

Cree un proyecto nuevo en un directorio llamado *tutorial2\multilevel,* y utilice el nombre *example_vhdl2* tanto para el proyecto como para la entidad de nivel superior. Seleccione la familia Cyclone y deje que el compilador elija un dispositivo específico.

Cree un archivo de diseño en VHDL llamado *example_vhdl2* que contenga el código de la figura 4.48, como se muestra en la figura C.9*a.* Compile el proyecto. Después de una compila-

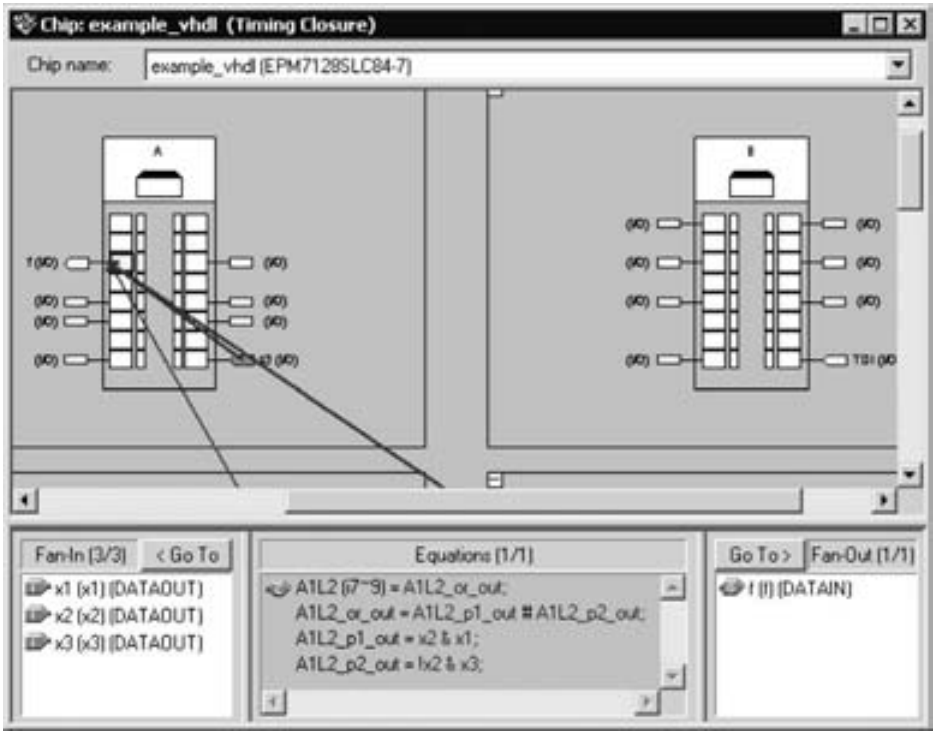


Figura C.7 Vista del factor de entrada de carga (*fan in*) y las ecuaciones del nodo.

ción exitosa, en el informe de compilación amplíe la sección Fitter y haga clic en Fitter Equations. En la parte inferior de esta sección la salida *f* está especificada como

$$\mathsf{f = OUTPUT(A1L3);}$$

Como se muestra en la figura C.9*b* la expresión lógica para A1L3 implementa *f* en una forma lógica multinivel. El primer nivel de lógica se especifica como

$$A1L2 = x_1(x_7x_2 + \bar{x}_6) + \bar{x}_1x_7x_2$$

Mostramos en el apéndice E que la celda lógica del FPGA Cyclone es una tabla de consulta (LUT) de cuatro entradas que puede implementar cualquier función de cuatro entradas. Como la expresión anterior tiene tal número de entradas, puede producirse en una celda lógica en el dispositivo. Esta celda proporciona una entrada a la expresión del siguiente nivel

$$A1L3 = A1L2(x_3 + x_4x_5)$$

Esta expresión también tiene cuatro entradas y, por tanto, puede producirse en una sola celda. De esta manera, *f* se implementa como dos celdas lógicas en cascada. Se anima al lector a comprobar que la expresión para A1L3 implementa en forma apropiada la función especificada en la figura C.9*a*.

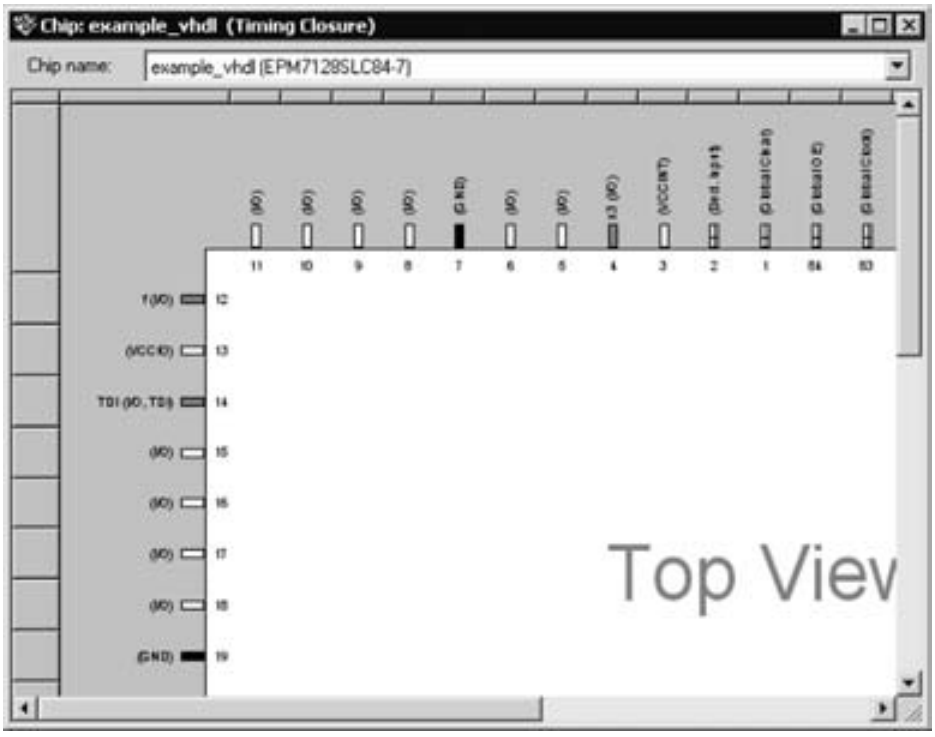
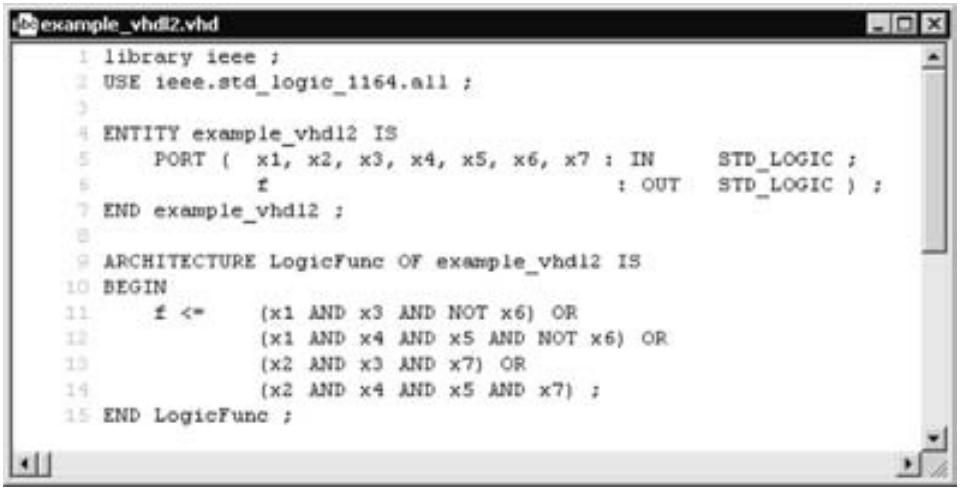


Figura C.8 Vista superior del encapsulado.

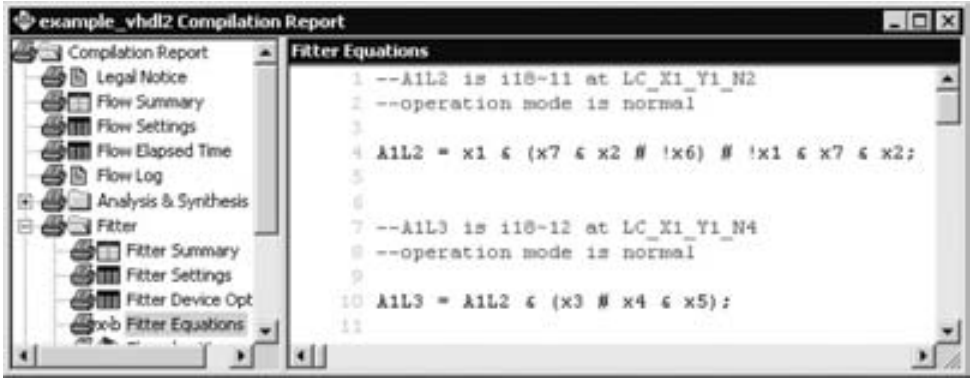
Después de implementar el diseño en el chip de Cyclone, realice una simulación de tiempo (como se explicó en la sección C.1.3) para formarse una idea de las características de sincronización del dispositivo de Cyclone. Una vez que un proyecto se ha compilado para el dispositivo objetivo, puede descargarse en un chip con Quartus II. El procedimiento para programar un chip se describe en el apéndice D.

C.3 IMPLEMENTACIÓN DE UN SUMADOR CON QUARTUS II

En la sección 5.5 mostramos cómo un sumador de acarreo en cascada de *n* bits puede especificarse en código de VHDL. En esta sección ilustramos cómo ese sumador puede implementarse con el sistema Quartus II. Cree un proyecto nuevo, *adder16,* en el directorio *tutorial2\addern.* Implementaremos el circuito sumador en un FPGA de Cyclone. Por consiguiente, en la ventana *New Project Wizard* que aparece en la figura B.7, seleccione la familia Cyclone. Elija Yes bajo la pregunta Do you want to assign a specific device y haga clic en el botón Next. En la pantalla del asistente que se abre en seguida elija el dispositivo EP1C6F256C7 (si este dispositivo no



a) El código fuente de VHDL.



b) El informe de ecuaciones del *Fitter* (*Fitter Equations*).

Figura C.9 El código fuente de *example_vhdl2* y su implementación.

está en la lista, cambie el elemento Speed Grade en el cuadro Filter a Any). Utilizamos este dispositivo porque está disponible en el tablero de desarrollo Cubic Cyclonium proporcionado por Altera (visite altera.com).

C.3.1 EL CÓDIGO DEL SUMADOR DE ACARREO EN CASCADA

El código de VHDL para el sumador de *n* bits se proporciona en la figura C.10. Toma la señal de acarreo de entrada, *carryin,* más dos números de *n* bits, *X* y *Y,* como entradas y produce la suma

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder16 IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( carryin   : IN   STD_LOGIC ;
           X, Y      : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           S         : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           carryout  : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Structure OF adder16 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO n−1) ;
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    FA_0: fulladd PORT MAP ( carryin, X(0), Y(0), S(0), C(1) ) ;
    G_1: FOR i IN 1 TO n−2 GENERATE
        FA_i: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1)) ;
    END GENERATE ;
    FA_n: fulladd PORT MAP ( C(n−1), X(n−1), Y(n−1), S(n−1), carryout ) ;
END Structure ;
```

Figura C.10 El código de VHDL para un sumador de acarreo en cascada.

de salida de *n* bits, *S,* y la señal de acarreo de salida, *carryout.* El código utiliza el parámetro *n*, de modo que el sumador puede parametrizarse a fin de que funcione para cualquier valor de *n.* En este ejemplo, *n* se establece en 16. En el código el vector *C* se usa para representar los acarreos intermedios entre las etapas del sumador. Se utiliza un ciclo for para crear *n* sumadores completos que formen el sumador de acarreo en cascada.

Escriba el código de la figura C.10 en el editor de texto, como explicamos en la sección B.4.2, y guarde el archivo en el directorio *tutorial2\addern* con el nombre *adder16.vhd.* Como el código instancia una entidad de sumador completo, llamada *fulladd,* cree otro archivo de VHDL para este subcircuito. El código para el sumador completo se muestra en la figura 5.22. Compile el circuito. El informe de compilación se muestra en la figura C.11.

C.3.2 SIMULACIÓN DEL CIRCUITO

Para probar la exactitud del circuito, realizaremos una simulación de tiempo. Para abreviar sólo emplearemos algunos vectores de prueba, pero en una situación de diseño real se requerirían pruebas más extensas.

Abra la ventana del editor de formas de onda. Utilice Edit | End Time para establecer que la simulación buscada se ejecute de 0 a 250 ns. Elija que las líneas de la cuadrícula se coloquen

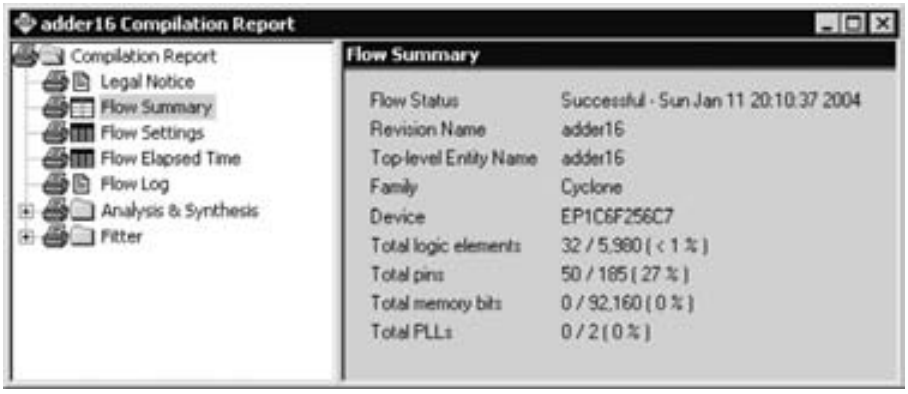


Figura C.11 El resumen del informe de compilación.

a intervalos de 25 ns. Esto se hace seleccionando Edit | Grid Size, lo que conduce a la ventana de la figura C.12. Establezca el periodo en 50 ns y haga clic en OK. Seleccione View | Fit in Window para exhibir todo el intervalo de simulación en la ventana.

Seleccione Edit | Insert Node or Bus y luego abra el programa auxiliar *Node Finder* para llegar a la ventana de la figura C.13. Establezca el filtro en Pins: all y haga clic en List, lo cual despliega los nodos de entrada y de salida como se representa en la figura. Desplácese hacia abajo por la lista de nodos hasta llegar a *carryin.* Seleccione este nodo haciendo clic sobre él y luego en el signo >. A continuación elija la entrada *X*. Nótese que esta entrada puede seleccionarse ya sea como nodos que corresponden a los bits individuales (indicados por medio de números encerrados en corchetes) o como un vector de 16 bits, lo que es una forma más práctica. Luego seleccione la entrada *Y* y las salidas *S* y *carryout.* Esto produce la imagen de la figura. Haga clic en OK.

Figura C.12 Establecimiento del espacio entre las líneas de la cuadrícula.

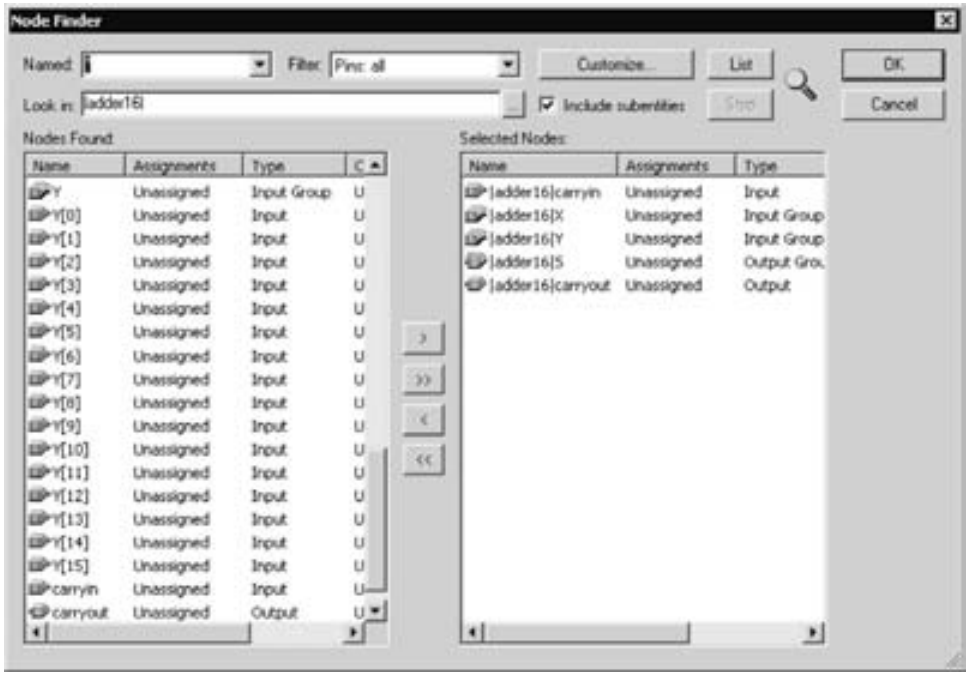


Figura C.13 La ventana del Node Finder.

La ventana del editor de formas de onda se parece ahora a la imagen de la figura C.14. Los vectores *X, Y* y *S* se trataron inicialmente como números binarios; también pueden tratarse como números octales, hexadecimales, o decimales con o sin signo. Para nuestro propósito es mejor tratarlos como hexadecimales, así que haga clic con el botón derecho del ratón sobre la *X* en la columna Name y seleccione Properties en el cuadro de lista desplegable para obtener la ventana mostrada en la figura C.15. Elija hexadecimal como la base, asegúrese que el ancho del bus sea 16 bits y haga clic en OK. (Quartus II utiliza el término *bus* para referirse a nodos multibits.) De igual forma, declare que *Y* y *S* deben tratarse como números hexadecimales. La pantalla de formas de onda resultante se muestra en la figura C.16.

Ahora estableceremos los valores de prueba de *X* y *Y.* El valor predeterminado de estas entradas es 0. Para asignar valores específicos en varios intervalos proceda como sigue. Seleccione (resalte) el intervalo de 100 a 175 ns de la entrada *X*. Presione el ícono de Arbitrary Value en la barra de herramientas (está etiquetado con un signo de interrogación) para abrir el cuadro de diálogo de la figura C.17. Introduzca el valor 3FFF y haga clic en OK. Luego establezca *X* en el valor 7FFF en el intervalo de 175 a 250 ns. Establezca *Y* en 0001 en el intervalo de 50 a 250 ns. Por tanto, las formas de onda de entrada deben ser como las de la figura C.18. Si éste fuera un proyecto de diseño real introduciríamos valores de prueba adicionales en las formas de onda, pero para propósitos de este tutorial unos cuantos vectores de prueba bastan. Guarde el archivo como *adder16.vwf.*

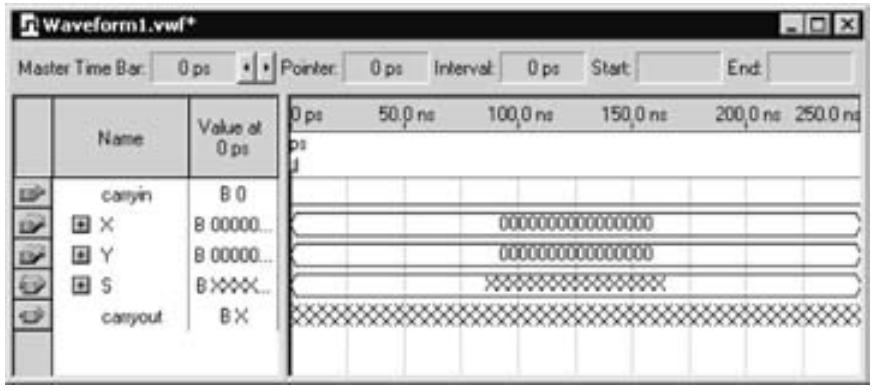


Figura C.14 Nodos de entrada y salida seleccionados.

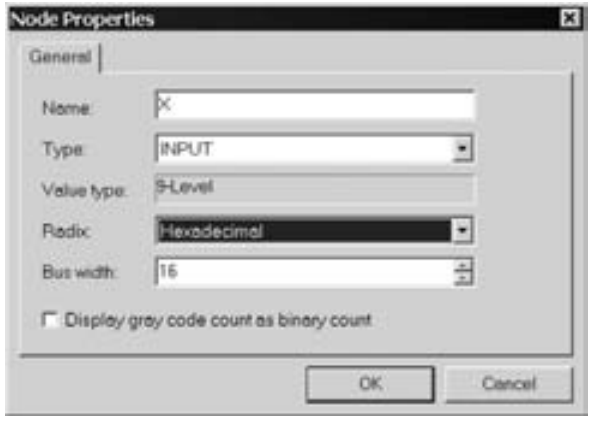


Figura C.15 Definición de las características de un nodo.

C.3.3 SIMULACIÓN DE TIEMPO

Para examinar la funcionalidad del circuito y determinar su velocidad de operación en el dispositivo elegido realizaremos una simulación de tiempo. Seleccione Assignments | Settings | Simulator para llegar a la ventana de la figura B.25 y elija Timing como el modo de simulación. Ejecute el simulador. El resultado se presenta en la figura C.18. Muestra retrasos considerables en la producción del valor correcto $S = 4000$ porque los acarreos son en cascada a través de las etapas del sumador.

Apunte a la manija de control en forma de un cuadrado pequeño en la parte superior de la línea de referencia y arrástrela al punto donde el valor *S* se vuelve 4000. Una vista más precisa puede obtenerse si la imagen de forma de onda se amplía con la herramienta *Zoom*. Agrande

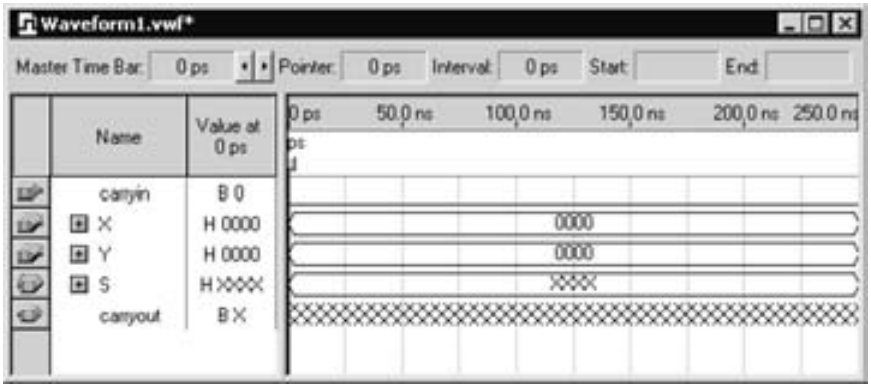


Figura C.16 Uso de la representación hexadecimal para señales multibit.

Figura C.17 Asignación del valor de una señal multibit.

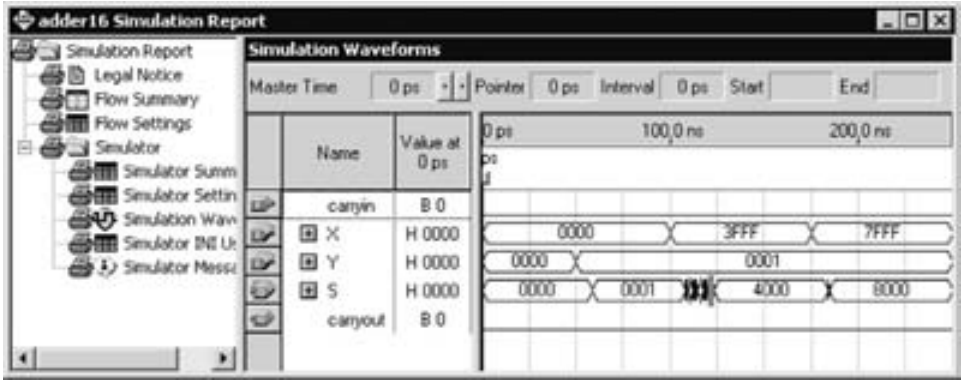


Figura C.18 El resultado de la simulación de tiempo.

esa imagen para que se vea como la pantalla de la figura C.19. Haga clic en el ícono de la herramienta de selección y arrastre la línea de referencia lo más cerca posible del punto donde el valor 4000 se vuelve válido.

El cambio en *S* de 0001 a 4000 es provocado por el cambio de la entrada *X* de 0000 a 3FFF, lo cual ocurre a los 100 ns. Como se observa en la figura C.19, la salida *S* cambia a 4000 aproximadamente a los 125.5 ns. Por consiguiente, el retraso de propagación por el sumador, para estos valores particulares de las entradas, se estima en 25.5 ns. Nótese que, en este caso, el sumador realiza la operación 3FFF + 1 = 4000, que supone un acarreo en cascada a través del grueso de las etapas del circuito sumador. Para otros valores de las entradas, el retraso de propagación puede ser mucho más pequeño. En la figura C.18 se advierte que la operación 0000 + 0001 = 0001 se completa en alrededor de 8.5 ns.

Cuando compilamos nuestro circuito utilizando Processing | Start Compilation uno de los módulos ejecutado es el *Timing Analyzer*. Como se explicó en el capítulo 12, este módulo produce automáticamente una estimación de la velocidad del circuito. Abra el informe de compilación seleccionando Processing | Compilation Report o haciendo clic en su icono. El informe incluye el análisis de tiempo derivado. Haga clic en el pequeño símbolo + que se halla junto al Timing Analyzer para ampliar esta sección del informe. Luego, haga clic en Timing Analyzer Summary para obtener la pantalla presentada en la figura C.20. El resumen indica que el retraso de propagación estimado en el peor de los casos desde un pin de entrada hasta un pin de salida, t_{pd}, es 24.7 ns. Esta ruta, la más larga, comienza en la entrada *carryin* y termina en *S*[15]. Nótese también que se estima que el retraso mínimo es de 8.5 ns. Información más detallada de los retrasos a lo largo de varias rutas por el circuito puede verse haciendo clic en tpd, en el lado izquierdo de la figura C.20, con lo que se muestra la información que aparece en la figura C.21. Aquí vemos que hay varias rutas a lo largo de las cuales el retraso de propagación está cercano al máximo, incluido aquel dado en el resumen de la figura C.20. Estas rutas del retraso más largo se conocen como *rutas críticas.*

El *Timing Analyzer* realiza varios tipos de análisis de tiempo. Los resultados presentados en la figura C.21 muestran los retrasos a través de un circuito combinacional, desde los pines de entrada hasta los de salida. Los otros tipos de análisis son aplicables sólo a circuitos que contienen elementos de almacenamiento, en concreto, flip-flops. Este tipo de análisis se estudia en la sección C.5.

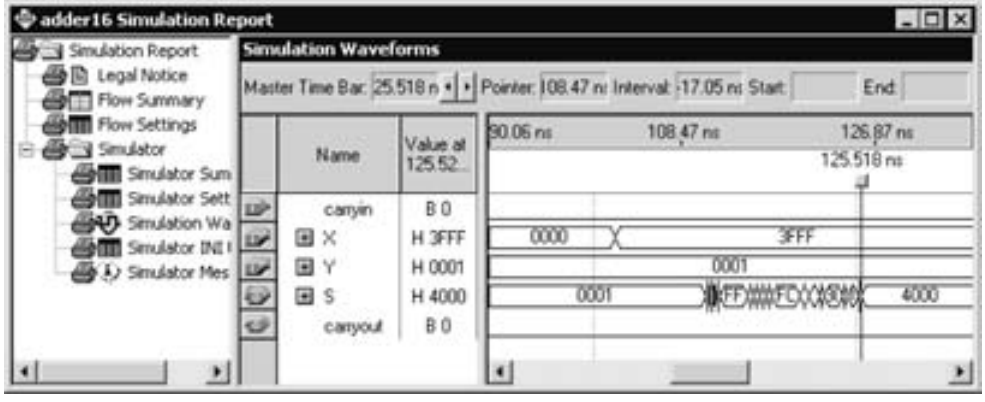


Figura C.19 Resultados detallados de la simulación de tiempo.

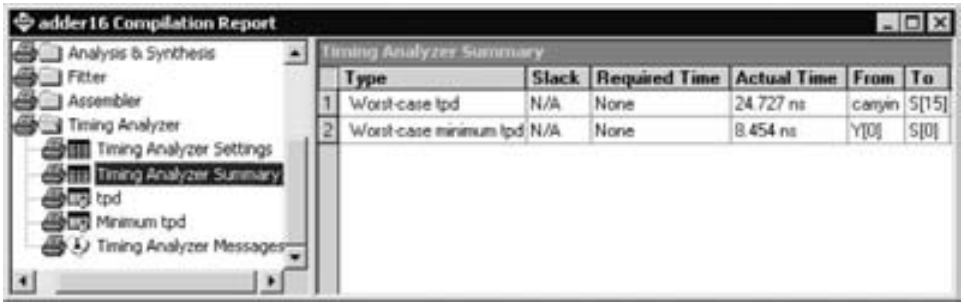


Figura C.20 El peor caso de retraso de propagación.

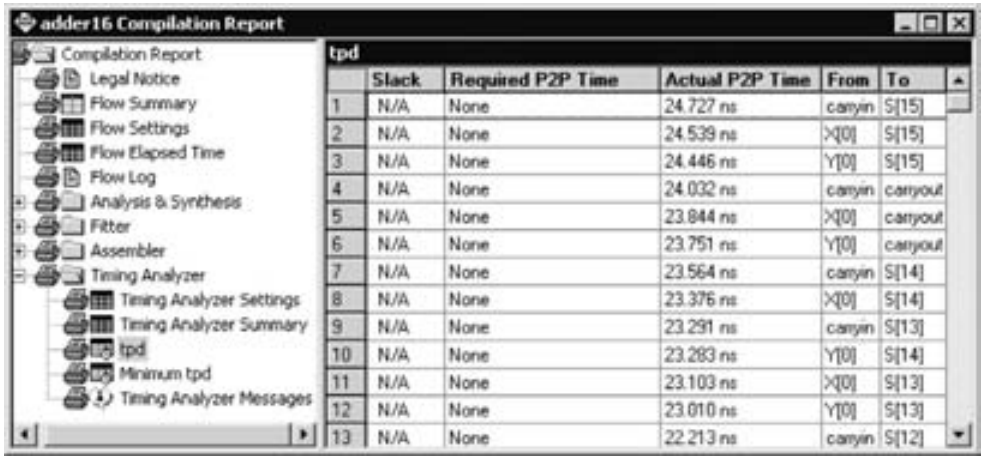


Figura C.21 Las rutas críticas.

C.3.4 IMPLEMENTACIÓN EN UN CHIP CPLD

Ahora implementaremos el circuito de acarreo en cascada en un chip CPLD. Seleccione Assignments | Device para llegar a la ventana de la figura C.22. Elija la familia MAX 7000S y seleccione el dispositivo EPM7128SLC84-7.

Compile el circuito. Abra el resumen del *Timing Analyzer* en el informe de compilación, el cual se muestra en la figura C.23. Nótese que el peor caso de retraso de propagación ahora es 22.5 ns, que es menor que el observado en la figura C.20. No debemos pasar a una conclusión sobre el rendimiento relativo de los dispositivos FPGA y CPLD, pues este circuito es sólo un ejemplo pequeño, y hay muchos otros dispositivos que podríamos haber elegido en nuestra implementación. Además, hay otras posibilidades en la implementación de un diseño, como veremos en la sección siguiente.

Hemos terminado de trabajar en el circuito *addern*, así que cierre el proyecto.

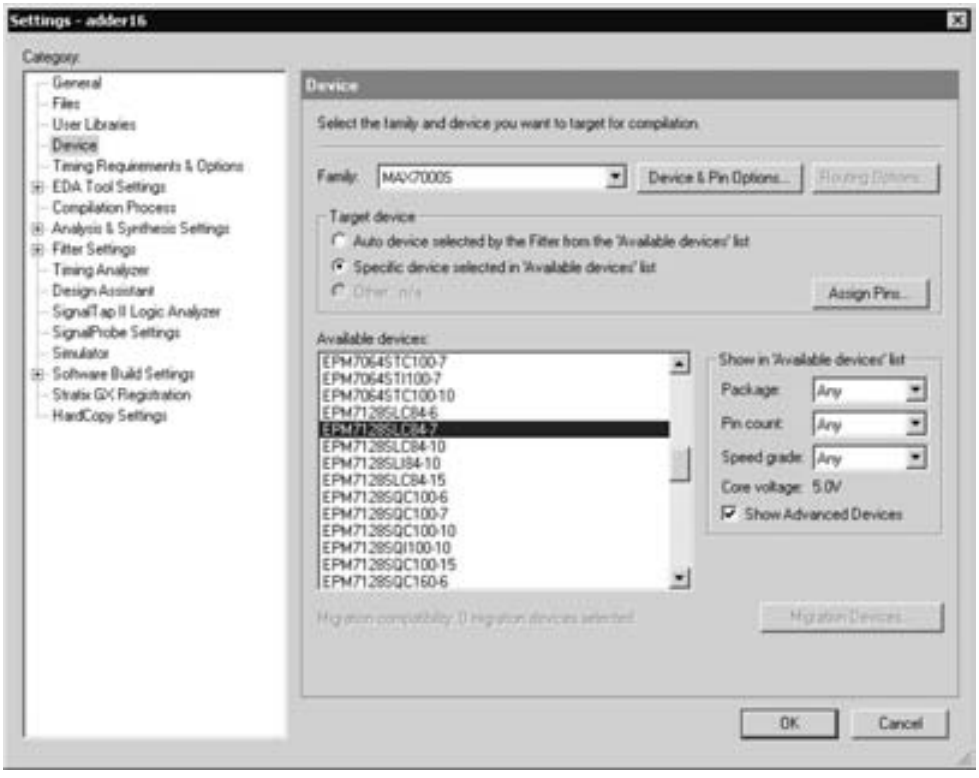


Figura C.22 Especificación del dispositivo buscado.

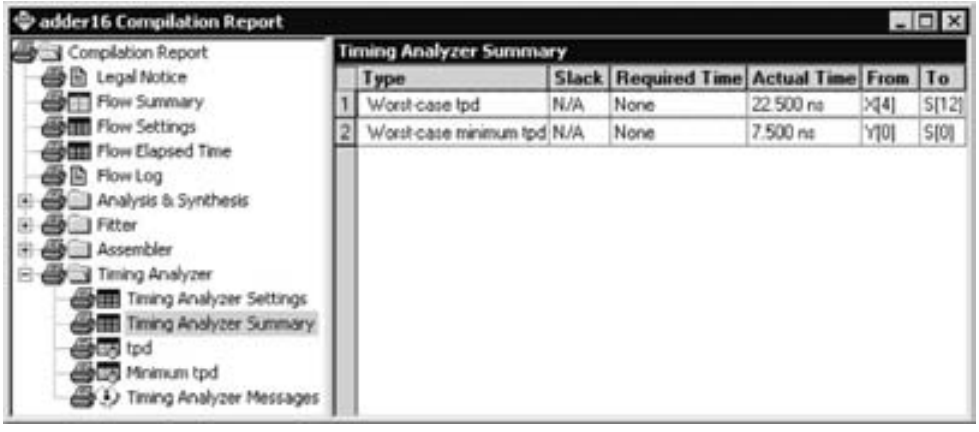


Figura C.23 El peor caso de retraso utilizando un CPLD.

C.4 USO DE UN MÓDULO LPM

En la sección 5.5.1 estudiamos cómo un circuito sumador puede implementarse con el módulo *lpm_add_sub* de la biblioteca de módulos parametrizados (LPM, *library of parameterized modules*). En esta sección comparamos el circuito sumador producido por el módulo *lpm_add_sub* con el sumador de acarreo en cascada implementado en la sección anterior. Cree un proyecto nuevo, *adder16_lpm,* en el directorio *tutorial2\adderlpm.* Elija el mismo chip FPGA que en la sección C.3.

La manera más fácil de instanciar un módulo LPM es por medio de un asistente. Seleccione Tools | MegaWizard Plug-in Manager para activar el asistente. Aparecerán varios cuadros de lista desplegables donde podemos especificar las funciones del módulo buscado. En la pantalla mostrada en la figura C.24 elija crear una nueva variación de una megafunción y luego haga clic en Next. En la pantalla de la figura C.25 seleccione el módulo LPM_ADD_SUB. Asegúrese de que la familia Cyclone está mostrada en la parte superior derecha, y también seleccione la entrada VHDL como el tipo de archivo por crear. Nombre el archivo de salida *megadd.vhd.* (La extensión del nombre de archivo, *vhd,* se añadirá automáticamente.) Haga clic en Next. En la figura C.26, especifique que se requiere un circuito sumador de 16 bits. Haga clic en Next para llegar a la pantalla de la figura C.27. Indique que ambas entradas pueden variar y haga clic en Next. En la figura C.28 especifique que son necesarias tanto la señal de acarreo de entrada como la de salida. Observe que el asistente muestra un símbolo para el sumador, el cual incluye las entradas y salidas especificadas. En la pantalla de la figura C.29 inhabilite la opción de encauzamiento (*pipelining*). La última pantalla se muestra en la figura C.30, la cual indica los archivos generados por el asistente. Haga clic en Finish. Como sólo nos interesa en el archivo *megadd.vhd*, asegúrese de que éste sea el único archivo seleccionado con una marca.

El módulo *megadd* se muestra en la figura C.31. (Hemos eliminado los comentarios para hacer la figura más pequeña.) El código de VHDL de nivel superior que instancia este módulo se presenta en la figura C.32. Introdúzcalo en un archivo llamado *adder16_lpm.vhd.*

Figura C.24 Eligiendo crear una instancia LPM.

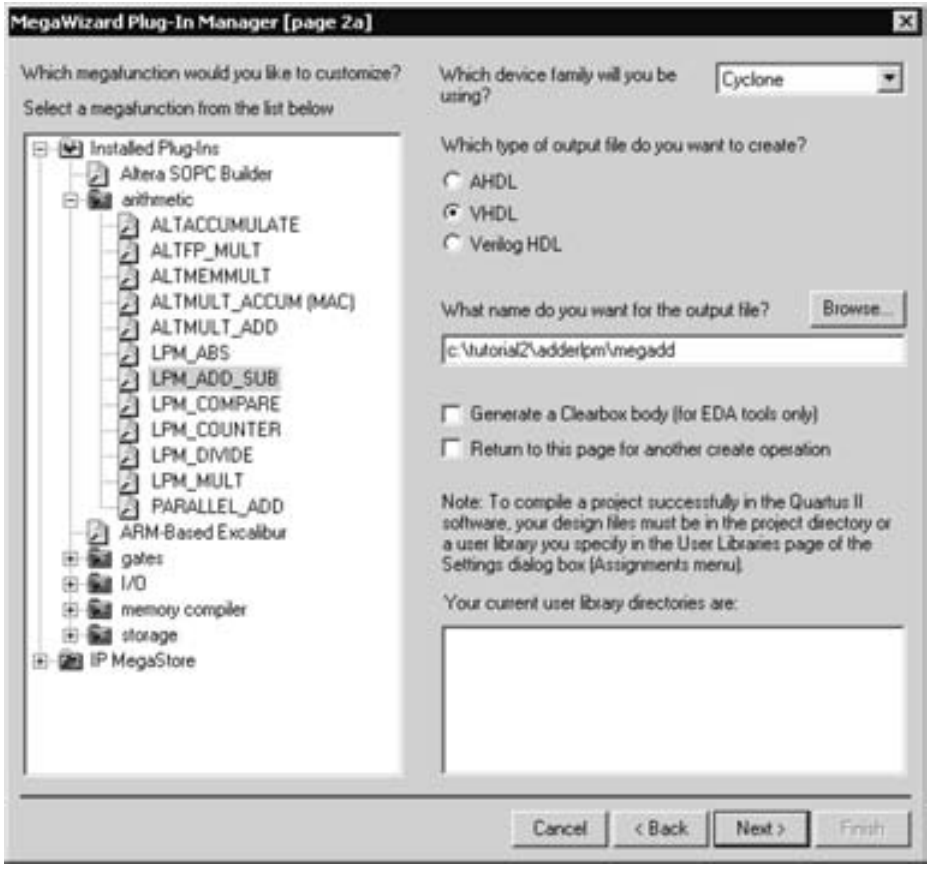


Figura C.25 Selección de la LPM y su especificación VHDL.

Compile el diseño. Un resumen del análisis de tiempo se muestra en la figura C.33. En este diseño, el retraso de propagación en el peor de los casos es de 13.2 ns. Desde luego, la implementación del sumador por medio de una LPM apropiada es superior a nuestra especificación genérica en la figura C.10. La razón de que este sumador sea mucho más rápido que nuestro sumador de acarreo en cascada creado antes es que la LPM utiliza sistemas de circuitos especiales en el FPGA para realizar la suma. Estudiamos este sistema de circuitos, a menudo llamado *cadena de acarreo,* en la sección 5.4. Podemos concluir que un diseñador normalmente debe usar una LPM si existe un módulo adecuado en la biblioteca. Cierre el proyecto *adder16_lpm*.

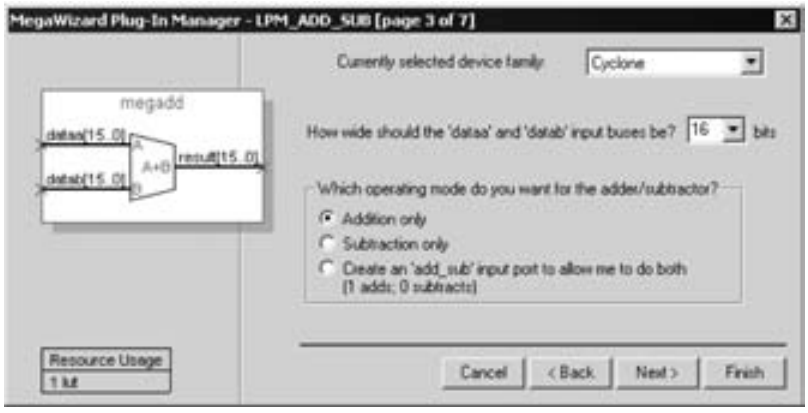


Figura C.26 Elección de la opción de sumador y el número de bits.

Figura C.27 Se desea que ambas entradas sean variables.

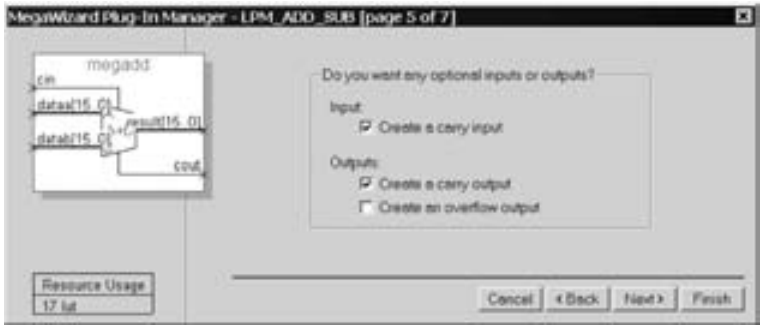


Figura C.28 Incluyendo las conexiones de acarreo de entrada y salida.

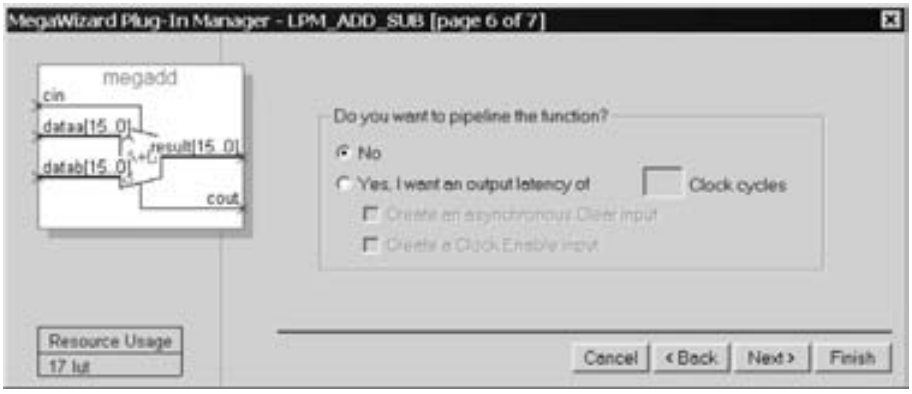


Figura C.29 Inhabilitando la opción de establecimiento de encauzamiento.

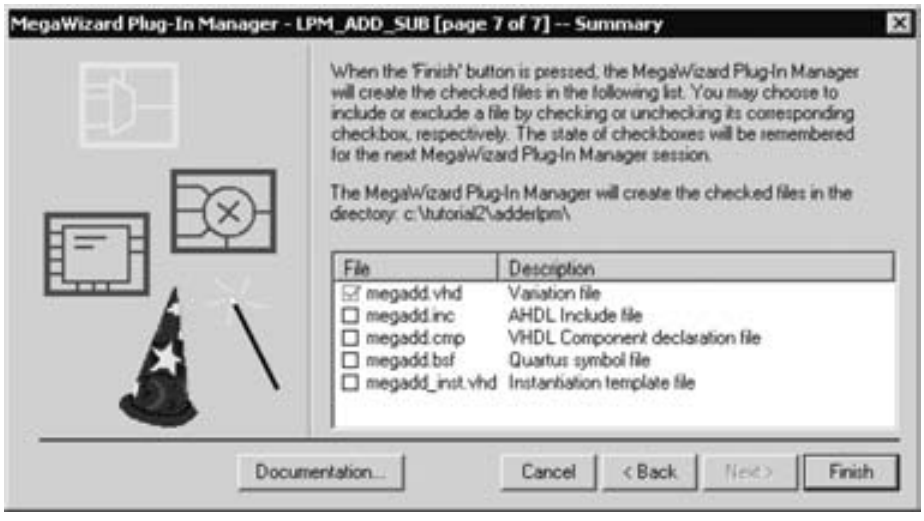


Figura C.30 Archivos generados por el asistente.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY megadd IS
    PORT ( dataa      : IN  STD_LOGIC_VECTOR (15 DOWNTO 0);
           datab      : IN  STD_LOGIC_VECTOR (15 DOWNTO 0);
           cin        : IN  STD_LOGIC ;
           result     : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
           cout       : OUT STD_LOGIC );
END megadd;

ARCHITECTURE SYN OF megadd IS
    SIGNAL sub_wire0 : STD_LOGIC ;
    SIGNAL sub_wire1 : STD_LOGIC_VECTOR (15 DOWNTO 0);

    COMPONENT lpm_add_sub
        GENERIC ( lpm_width     : NATURAL;
                  lpm_direction : STRING;
                  lpm_type      : STRING;
                  lpm_hint      : STRING );
        PORT ( dataa      : IN  STD_LOGIC_VECTOR (15 DOWNTO 0);
               datab      : IN  STD LOGIC VECTOR (15 DOWNTO 0);
               cin        : IN  STD_LOGIC ;
               cout       : OUT STD_LOGIC ;
               result     : OUT STD_LOGIC_VECTOR (15 DOWNTO 0) );
    END COMPONENT;
BEGIN
    cout <= sub_wire0;
    result <= sub_wire1(15 DOWNTO 0);

    lpm_add_sub_component : lpm_add_sub
        GENERIC MAP ( lpm_width => 16,
                      lpm_direction => "ADD",
                      lpm_type => "LPM_ADD_SUB",
                      lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=YES")
        PORT MAP ( dataa => dataa,
                   datab => datab,
                   cin => cin,
                   cout => sub_wire0,
                   result => sub_wire1 );
END SYN;
```

Figura C.31 Código de VHDL para el módulo *megadd*.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder16_lpm IS
    PORT ( carryin   : IN   STD_LOGIC ;
           X, Y      : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S         : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           carryout  : OUT  STD_LOGIC ) ;
END adder16_lpm ;

ARCHITECTURE Structure OF adder16_lpm IS
    COMPONENT megadd
        PORT ( dataa  : IN   STD_LOGIC_VECTOR (15 DOWNTO 0);
               datab  : IN   STD_LOGIC_VECTOR (15 DOWNTO 0);
               cin    : IN   STD_LOGIC ;
               result : OUT  STD_LOGIC_VECTOR (15 DOWNTO 0);
               cout   : OUT  STD_LOGIC ) ;
    END COMPONENT ;
BEGIN

    adder_circuit: megadd PORT MAP ( cin => carryin, dataa => X,
        datab => Y, result => S, cout => carryout ) ;

END Structure ;
```

Figura C.32 Código de VHDL que instancia el sumador LPM.

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To
1	Worst-case tpd	N/A	None	13.242 ns	Y[0]	S[15]
2	Worst-case minimum tpd	N/A	None	9.079 ns	X[0]	S[0]

Figura C.33 El peor caso de retraso para el circuito *adder16_lpm*.

C.5 DISEÑO DE UNA MÁQUINA DE ESTADO FINITO

En este ejemplo se muestra cómo implementar un circuito secuencial con Quartus II. La explicación presupone que el lector ha estudiado el capítulo 8. En la sección 8.1 mostramos una máquina de estado finito (FSM) simple tipo Moore, con una entrada, *w,* y una salida, *z.* Siempre que *w* es 1 para dos ciclos de reloj sucesivos, *z* se establece en 1. El diagrama de estado para la

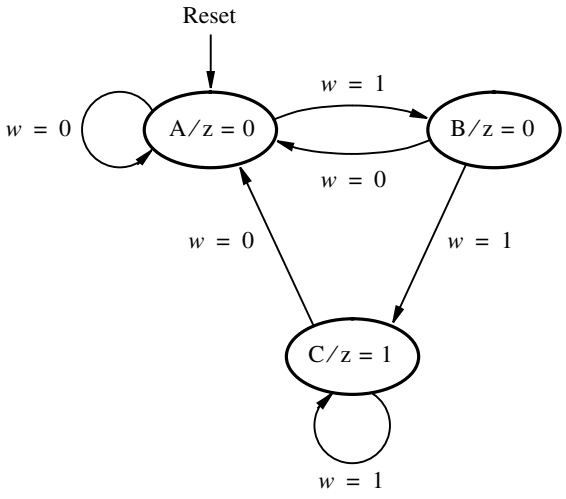


Figura C.34 Diagrama de estado de una FSM tipo Moore.

FSM se presenta en la figura 8.3, se reproduce en la figura C.34. El código de VHDL que describe la máquina aparece en la figura 8.33; se reproduce en la figura C.35. Cree un proyecto nuevo, *simple,* en el directorio *tutorial2\fsm.* Cree un archivo nuevo en el editor de texto e introduzca el código mostrado en la figura C.35. Guarde el archivo con el nombre *simple.vhd.*

C.5.1 IMPLEMENTACIÓN EN UN CPLD

Seleccione el mismo dispositivo MAX 7000S que en la sección C.1. Compile el circuito. Abra el editor de formas de onda e importe los nodos *Resetn, Clock, w* y *z*. Estos nodos se encuentran estableciendo el filtro del *Node Finder* en Pins: all. También queremos ver el comportamiento de las variables de estado, las cuales se implementan por medio de flip-flops. Para encontrar estos nodos, establezca el filtro del *Node Finder* en Registers: post-fitting y haga clic en List. El *Node Finder* despliega dos nodos, como se muestra en la figura C.36. Impórtelos al editor de formas de onda. Fije el tiempo de simulación total en 650 ns y establezca el tamaño de la cuadrícula en 25 ns. Establezca *Resetn* = 0 durante los primeros 50 ns, y luego en 1. Para introducir la forma de onda para la señal de reloj, haga clic en el nombre de la forma de onda *Clock* en la pantalla del editor de formas de onda. Con la señal resaltada, haga clic en el icono *Overwrite Clock* de la barra de herramientas, el cual muestra un reloj. Con ello se abre el cuadro de diálogo mostrado en la figura C.37. Fije el periodo del reloj en 50 ns; asegúrese de que la fase es 0 y el ciclo de trabajo es de 50%; luego haga clic en OK. La señal de reloj definida ahora se muestra en la ventana del editor de formas de onda, como aparece en la figura C.38. A continuación, trace la forma de onda para *w* como se indica en la figura. Guarde el archivo con el nombre *simple.vwf.* Ejecute el simulador de tiempo para obtener el resultado mostrado en la figura C.39.

La FSM se comporta correctamente estableciendo $z = 1$ en cada ciclo del reloj para el cual $w = 1$ en los dos ciclos de reloj anteriores. Examine los retrasos de tiempo en el circuito, uti-

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY simple IS
    PORT ( Clock, Resetn  : IN   STD_LOGIC ;
           w              : IN   STD_LOGIC ;
           z              : OUT  STD_LOGIC ) ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    TYPE STATE_TYPE IS (A, B, C) ;
    SIGNAL y_present, y_next : STATE_TYPE ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = 0 ' THEN y_next <= A ;
                ELSE y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figura C.35 Código de VHDL para la FSM de la figura C.34.

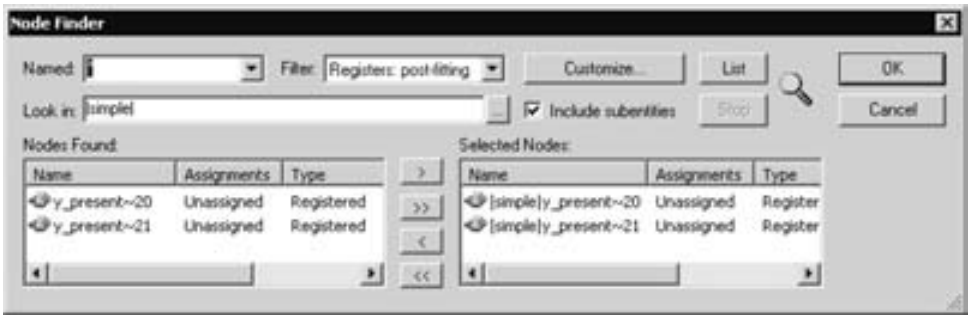


Figura C.36 Nodos que representan las variables de estado.

Figura C.37 Creación de la forma de onda para el reloj, *Clock*.

lizando la línea de referencia en el editor de formas de onda. Obsérvese que los cambios en el estado de la FSM ocurren 2.5 ns después de un flanco activo del reloj y que se necesitan 4.5 ns para cambiar el valor de la salida *f*.

Abra el resumen del analizador de tiempo en el informe de compilación, el cual aparece en la figura C.40. La fila inferior indica que la frecuencia máxima, que a menudo se llama *fmax,* en la que el circuito sintetizado puede operar es 125 MHz. Éste es un indicador de rendimiento útil. La f_{max} queda determinada por el retraso de propagación más largo entre dos registros (flip-flops). En la figura también se muestran los valores de algunos otros parámetros. Asimismo se proporcionan el tiempo de preparación del flip-flop del peor caso, t_{su}, y el tiempo de espera, t_h. La línea 1 de la figura C.40 especifica que la entrada *w* no puede cambiar en los 6 ns del flanco activo del reloj, ya que de lo contrario el flip-flop *y_present*~21 puede volverse inestable. La línea 3 muestra que ninguna señal de entrada en nuestro circuito tiene que permanecer estable después del flanco activo del reloj, pues el requisito de tiempo de espera del peor caso es negativo.

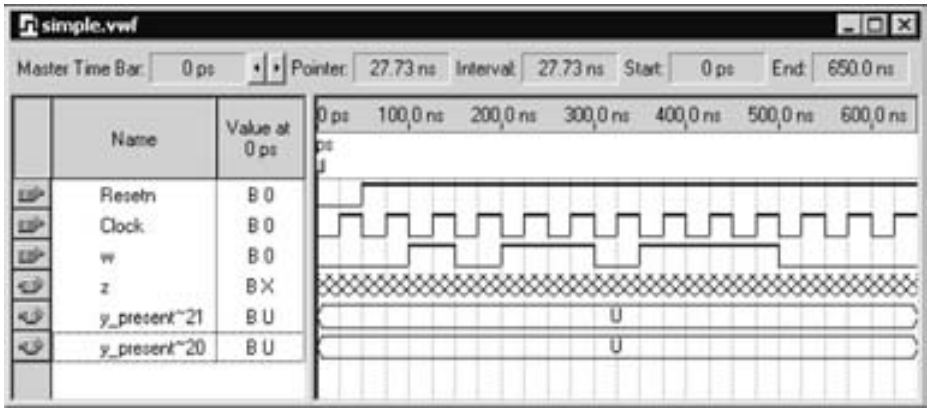


Figura C.38 Vectores de prueba de entrada.

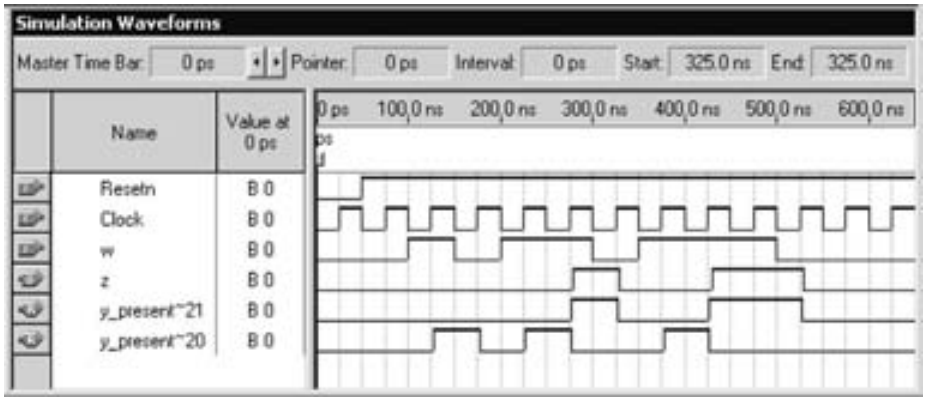


Figura C.39 Formas de onda de la simulación de tiempo.

En la sección 10.3.2 explicamos cómo se determinan los parámetros de tiempo del flip-flop en el chip objetivo. El parámetro t_{co} indica el tiempo transcurrido desde un flanco activo de la señal del reloj en el pin del reloj hasta que una señal de salida se produce en un pin de salida. Ese retraso es de 4.5 ns para la salida *z*, que es lo que observamos también en las formas de onda de la figura C.39.

Nótese que los estados de esta FSM se implementaron utilizando dos variables de estado. El código de VHDL de la figura C.35 especificó las variables de estado actual como *y_present*[1] y *y_present*[2]. Sin embargo, Quartus II las nombró *y_present~20* y *y_present~21*, como descubrimos cuando usamos el *Node Finder*. Quartus II emplea los nombres de todas las entradas y salidas tal como se proporcionan en el código de VHDL, pero puede generar nombres distintos para conexiones internas.

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To
1	Worst-case tsu	N/A	None	6.000 ns	w	y_present~21
2	Worst-case tco	N/A	None	4.500 ns	y_present~21	z
3	Worst-case th	N/A	None	-1.000 ns	w	y_present~21
4	Worst-case minimum tco	N/A	None	4.500 ns	y_present~21	z
5	Clock Setup: 'Clock'	N/A	None	125.00 MHz (period = 8.000 ns)	y_present~21	y_present~20

Figura C.40 Resumen del análisis de tiempo para el circuito de la FSM.

Figura C.41 Agrupación de señales.

Dos o más señales binarias mostradas en el editor de formas de onda pueden combinarse en un "grupo" (que corresponde a un vector en terminología de VHDL) de señales, que pueden referirse por medio de un solo nombre. Abra el archivo *simple.vwf* y seleccione *y_present~21* y *y_present~20* al mismo tiempo, de modo que sus formas de onda se resalten (asegúrese de que *y_present~21* se enumera arriba de *y_present~20,* como se muestra en la figura C.38). Seleccione Edit | Group para abrir el cuadro de diálogo de la figura C.41. Escriba *y* como el nombre de grupo, elija hexadecimal como la base y haga clic en OK. Esto hace que se use *y* en vez de *y_present~21* y *y_present~20* en el archivo *simple.vwf.* Realice la simulación de tiempo para obtener el resultado de la figura C.42. Ahora, los estados de la FSM se representan por medio de los valores del vector *y*.

C.5.2 IMPLEMENTACIÓN EN UN FPGA

En la sección 8.8 dijimos que cuando se implementa una FSM en un FPGA, una buena estrategia consiste en utilizar la codificación de uno activo, con una variable de estado asignada a cada estado. La herramienta de síntesis de Quartus II elige automáticamente ese método cuando opera para un chip FPGA específico.

Se invita al lector a recompilar el código *simple.vhd* para el mismo chip FPGA utilizado en la sección C.3. Hágalo y observe que se utilizan tres flip-flops para implementar la FSM. Los resultados del análisis de tiempo deben mostrar que el circuito operará a una $f_{máx}$ de alrededor de 320 MHz.

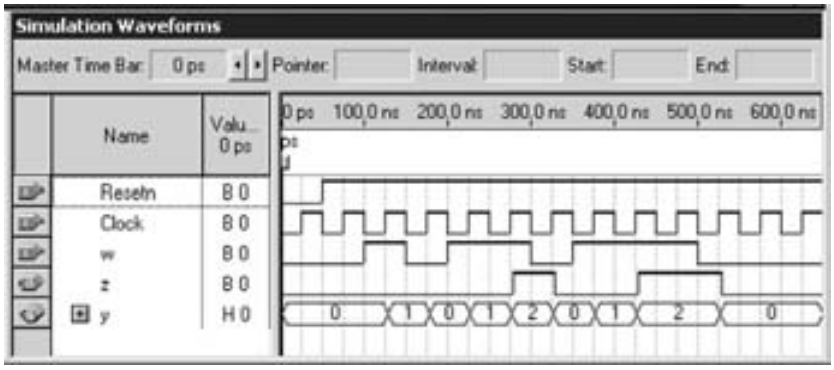


Figura C.42 Forma de onda desplegada como un vector *y*.

C.6 COMENTARIOS FINALES

Al haber completado este tutorial y el anterior, el lector conoce una buena parte de las funciones más importantes de Quartus II. En el tutorial siguiente mostraremos cómo puede manipular cuáles pines del chip objetivo se utilizan en un circuito, y cómo se realiza la programación de PLD con Quartus II.

apéndice

D

TUTORIAL 3
IMPLEMENTACIÓN FÍSICA EN UN PLD

En este tutorial nos centramos en la implementación física de un proyecto de diseño en un dispositivo específico. Mostramos cómo elegir manualmente cuáles pines de un encapsulado se utilizan para las señales de entrada y de salida en un circuito, y describimos cómo utilizar el módulo *Programmer* de Quartus II para transferir el proyecto de diseño compilado al chip PLD elegido.

D.1 ASIGNACIONES DE PINES

En los ejemplos del tutorial 2, la asignación de señales para los pines del dispositivo la realizó en forma automática el compilador (*Compiler*). En algunos casos el diseñador debe especificar manualmente qué pines se han de utilizar para ciertas señales de un circuito. Por ejemplo, la tarjeta de circuito que contiene el o los chips que están usándose puede tener conexiones alambradas de algunos de los pines del dispositivo a otros componentes, como interruptores o LED. Para emplear las conexiones alambradas, el diseñador debe especificar a qué señales de pines de dispositivo se asignarán.

En la sección C.1.4 describimos cómo examinar los resultados de compilación mediante la herramienta *Floorplan Editor* (editor de pines). En la figura C.8 se presentó la vista superior del encapsulado, así como las asignaciones de las señales *x*3 y *f* a los pines 4 y 12, respectivamente. En la sección D.1.3 mostraremos cómo las asignaciones de pines pueden modificarse con el editor de pines. Quartus II tiene varias formas de hacer asignaciones de pines; primero describiremos un método que utiliza el cuadro de diálogo Assignments.

Para asignar los pines manualmente es necesario especificar cuál chip se va a utilizar. Ya lo hicimos en la sección C.1.1, cuando seleccionamos el EPM7128SLC84-7, como se muestra en la figura C.2. Abra de nuevo el proyecto *example_vhdl,* creado en la sección C.1. Seleccione Assignments | Assign Pins para abrir la ventana de la figura D.1. El cuadro etiquetado Available Pins & Existing Assignments enumera todos los pines del dispositivo y muestra las asignaciones después que se han hecho. Como ejemplo, asignaremos las entradas *x*1, *x*2 y *x*3 a los pines 9, 10 y 11. Según se indica en la figura D.1, desplácese hacia abajo hasta que estos pines estén visibles

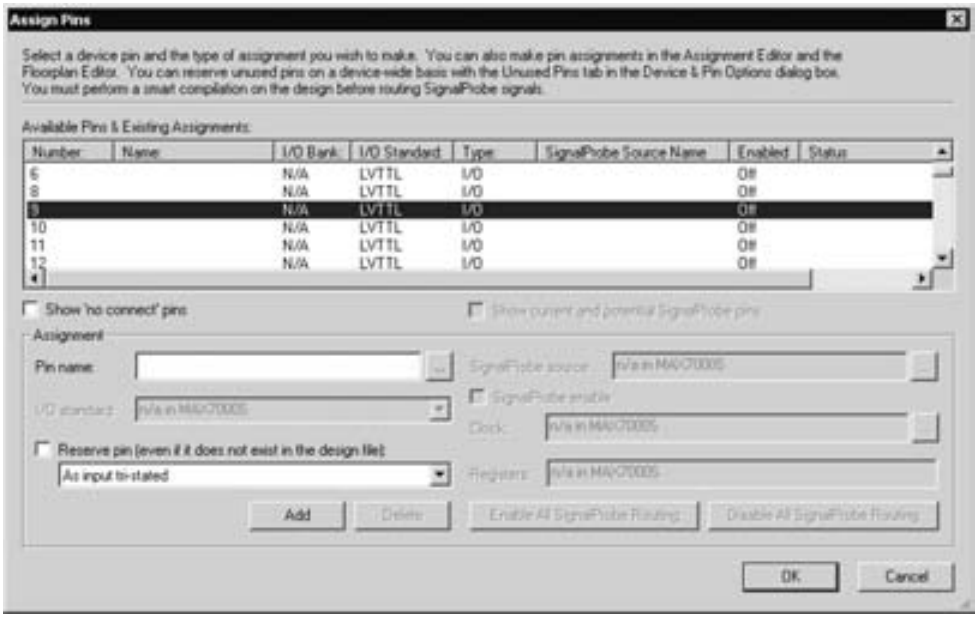


Figura D.1 Cuadro de diálogo para la asignación de pines (*Assign Pins*).

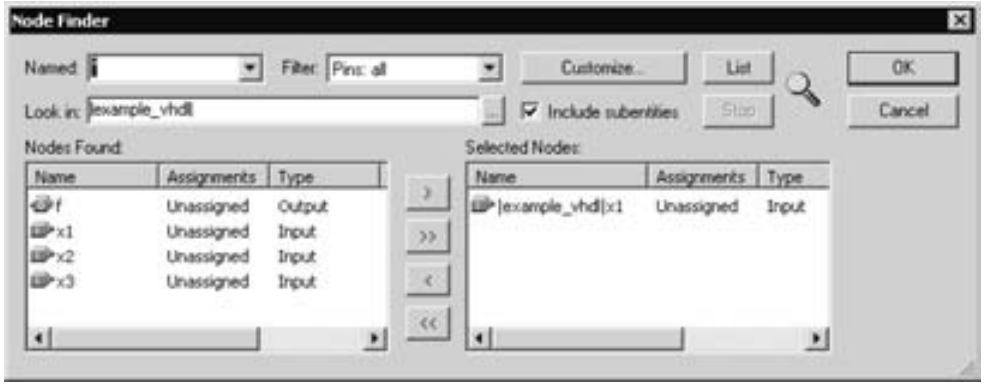


Figura D.2 Ventana del *Node Finder*.

y haga clic en el pin 9 para resaltarlo. En la sección Assignment haga clic en el botón con puntos suspensivos junto al cuadro Pin name. Con ello se abre la ventana *Node Finder* mostrada en la figura D.2.

Establezca Filter en Pins: all y haga clic en List para buscar los pines. Haga clic en el pin de entrada $x1$ y luego en el botón $>$ para mover este pin al cuadro Selected Nodes. Ahora haga clic en OK para regresar al cuadro de diálogo Assign Pins. En la ventana de la figura D.1 haga

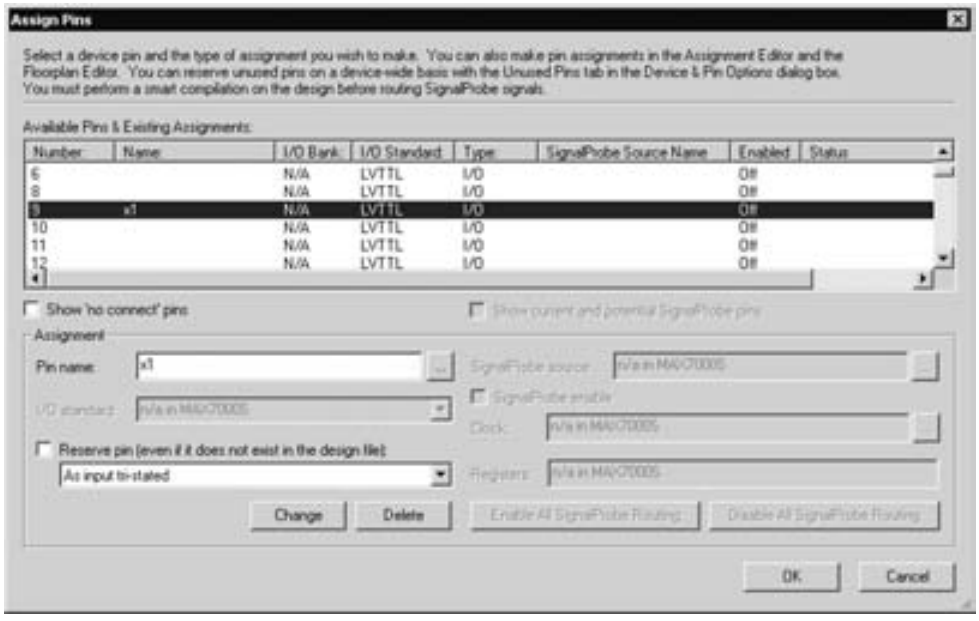


Figura D.3 Asignación de la entrada *x*1 al pin 9.

Number:	Name:	I/O Bank:	I/O Standard:	Type:	SignalProbe Source Name:
6		N/A	LVTTL	I/O	
8		N/A	LVTTL	I/O	
9	x1	N/A	LVTTL	I/O	
10	x2	N/A	LVTTL	I/O	
11	x3	N/A	LVTTL	I/O	
12		N/A	LVTTL	I/O	

Figura D.4 Asignación de pines para las entradas *x*1, *x*2 y *x*3.

clic en el botón Add. Esta acción hace que *x*1 aparezca en la columna Name al lado del pin 9 y sustituye el botón Add con un botón Change, como se presenta en la figura D.3.

Siguiendo el mismo procedimiento, haga clic en el pin 10 y asigne la entrada *x*2, luego haga clic en el pin 11 y asigne la entrada *x*3. El cuadro Available Pins & Existing Assignments ahora debe mostrar las asignaciones dadas en la figura D.4. Haga clic en OK para cerrar la ventana Assign Pins. Ahora el cuadro de diálogo Assignments es la ventana activa en la pantalla de Quartus II. Aun cuando no abrimos este cuadro de diálogo directamente, se abrió de manera implícita cuando seleccionamos el comando Assign Pins. Para completar la asignación de pines debe hacer clic en OK para cerrar la ventana Assignments. Si hace clic en Cancel para cerrarla se descartarán las asignaciones de los pines.

Como no hemos recompilado el proyecto *example_vhdl*, los resultados de la compilación aún no se han visto afectados por nuestra asignación de pines. En este punto, Quartus II almacena internamente las asignaciones; cuando el proyecto se cierra, éstas se guardan permanentemente en un archivo con extensión .qsf, formada por las siglas en inglés de *archivo de configuración de Quartus.* Seleccione File | Save Project para que Quartus II actualice este archivo y luego elija File | Open para examinar el archivo *example_vhdl.qsf* en el editor de texto. Desplácese por este archivo o utilice Edit | Find para localizar las tres asignaciones de pines, las cuales tienen la forma

```
set_location_assignment Pin_9   -to x1
set_location_assignment Pin_10 -to x2
set_location_assignment Pin_11 -to x3
```

Puede modificar, añadir o eliminar asignaciones de pines si edita este archivo, pero no es recomendable hacerlo porque es fácil cometer un error de sintaxis.

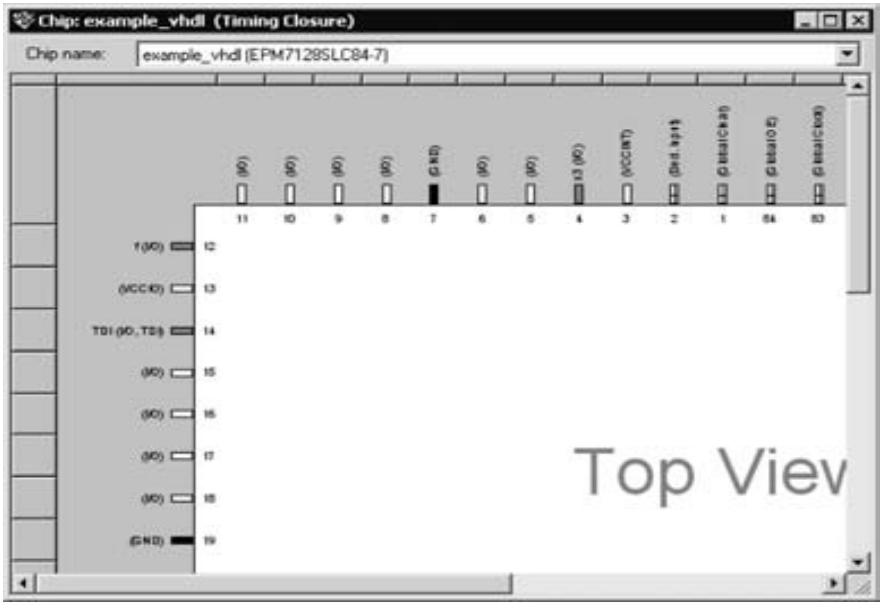
D.1.1 ANÁLISIS DE LAS ASIGNACIONES DE PINES CON EL EDITOR DE PINES

Como mencionamos, es posible ver la asignación de pines tras una compilación usando el *Floorplan Editor*. Quartus II también proporciona una herramienta de editor de pines que puede mostrar tanto las asignaciones de pines producidas desde la última compilación como las del usuario que no se han compilado aún. Seleccione Assignments | Timing Closure Floorplan para abrir la herramienta *Floorplan Editor* y desplegar la ventana de la figura D.5. Si no tiene la vista del encapsulado seleccionada, haga clic en View | Package Top.

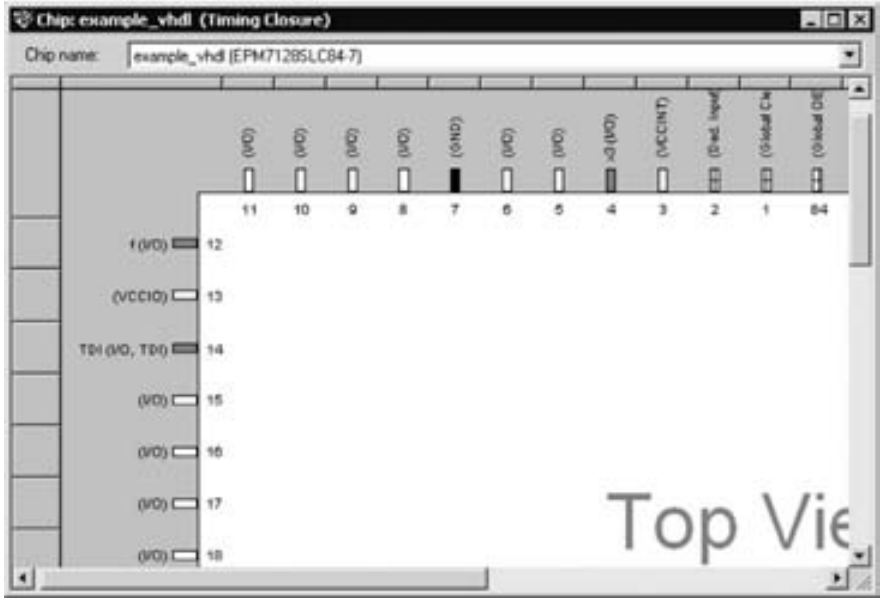
Bajo el menú View | Assignments hay dos opciones: Show User Assignments y Show Fitter Placements. Si la primera opción está activa, entonces se muestran las asignaciones del usuario, como las asignaciones de pines hechas en la sección D.1; si la segunda opción está habilitada, entonces aparecen los resultados de la última compilación. Como las dos opciones son independientes, puede elegir ver ambos tipos de asignaciones, una a la vez o ninguna. En la figura D.5*a* activamos sólo la opción View | Assignments | Show Fitter Placements, la cual genera la misma imagen mostrada en la figura C.8. En la figura D.*5b* se utiliza la opción View | Assignments | Show User Assignments, así que se presentan las asignaciones de pines que hicimos para las entradas *x*1, *x*2 y *x*3. Cada tipo de asignación también puede mostrarse en la vista del dispositivo, en vez en una vista del encapsulado. Seleccione View | Interior Cells para experimentar con esta función.

D.1.2 RECOMPILACIÓN DEL PROYECTO CON ASIGNACIONES DE PINES

Para cambiar los resultados de la compilación usando nuestras asignaciones de pines, vuelva a compilar el proyecto. Durante el proceso de compilación el *Fitter* utiliza las asignaciones de pines para las señales que se han especificado manualmente y hace asignaciones de pines automáticas para otras señales. La herramienta *Floorplan Editor* ahora debe mostrar las nuevas asignaciones de pines cuando se seleccione View | Assignments | Show Fitter Placements.



a) Vista del *Fitter Placement*.



b) Vista de las asignaciones del usuario.

Figura D.5 Vista de las asignaciones de pines en el editor de pines.

D.1.3 CÓMO CAMBIAR LAS ASIGNACIONES DE PINES CON EL FLOORPLAN EDITOR

Explicamos al principio de la sección D.1 cómo hacer asignaciones de pines con el comando Assignments | Assign Pins. Otra forma de crear, o cambiar, las asignaciones de pines es utilizando el editor de pines. Si no lo ha hecho, seleccione Assignments | Timing Closure Floorplan y haga clic en View | Package Top. Para ver tanto los resultados de la compilación como las asignaciones del usuario al mismo tiempo active View | Assignments | Show Fitter Placements y View | Assignments | Show User Assignments. Haga clic en el pin para la salida *f,* arrástrelo con el ratón y suéltelo en el pin 15. La pantalla de planos debe ser similar a la de la figura D.6. Con esta operación se crea una nueva asignación de pines para *f*. Usted puede seleccionar File | Save Project y luego utilizar el editor de textos para ver la asignación de pines cambiada en el archivo *example_vhdl.qsf.*

Cualquier asignación de pines puede modificarse de esta forma empleando el editor de pines. Otra variante para hacer asignaciones de pines consiste en abrir la herramienta *Node Finder* para buscar pines y luego utilizar el ratón para arrastrar y soltar los nombres de pines en los pines del encapsulado en el editor de pines. Las nuevas asignaciones que se crean sólo afectan los resultados de la compilación cuando el proyecto vuelve a compilarse.

Las asignaciones de pines pueden eliminarse de un proyecto con las mismas herramientas empleadas para crearlas. En el cuadro de diálogo Assign Pins, resalte con un clic una asignación de pines existente y luego haga clic en el botón Delete (figura D.3). En el editor de pines haga clic

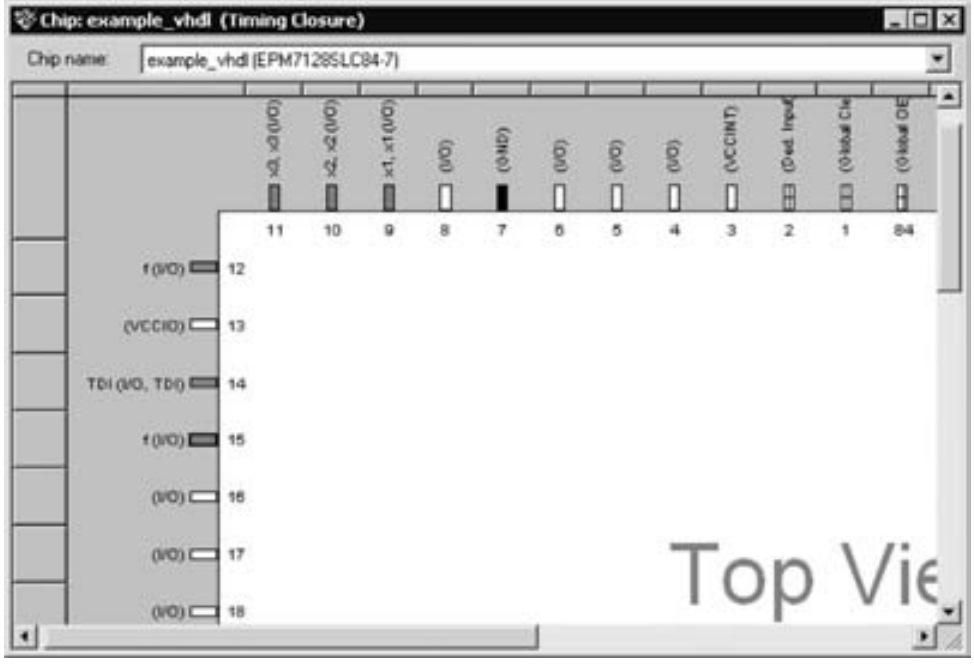


Figura D.6 Uso del Floorplan Editor para hacer asignaciones de pines.

en un pin que tiene una asignación de pines y luego en Edit | Delete para eliminarla. De nuevo, estos cambios sólo afectan los resultados de la compilación cuando el proyecto se recompila.

D.2 DESCARGA DE UN CIRCUITO EN UN DISPOSITIVO

Una vez que se ha compilado un circuito, puede descargarse en el dispositivo seleccionado. La descarga implica la programación de los interruptores apropiados en el chip para implementar el circuito deseado. Para ilustrar los pasos que ello supone, describiremos cómo se descarga un circuito en una tarjeta de desarrollo de laboratorio que se consigue con Altera Corporation. La tarjeta se llama UP-1 Education Board e incluye tanto un CPLD MAX 7000 CPLD como un FPGA. La tarjeta UP-1 puede obtenerse siguiendo las instrucciones de la sección University Program en el sitio web de Altera, en http://www.altera.com.

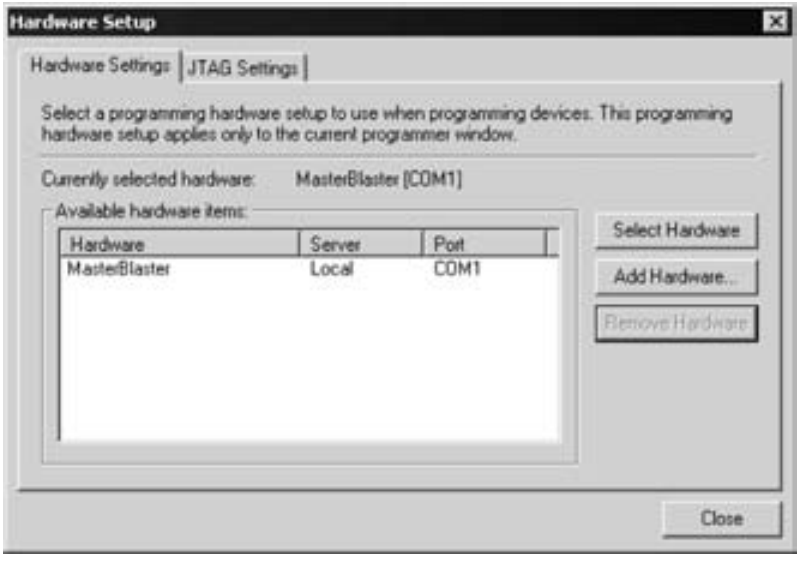
Describiremos cómo el proyecto *example_vhdl* que implementamos en un CPLD MAX 7000 puede descargarse en la tarjeta UP-1, suponiendo que está conectado a la computadora del lector. Un lector que no tenga acceso a la tarjeta UP-1 no podrá descargar el circuito, pero los pasos seguirán siendo fáciles de seguir. La tarjeta UP-1 se conecta a la computadora con un tipo de cable que vende Altera. Para propósitos de la explicación supondremos que se usa un cable ByteBlaster, que proporciona una conexión a un puerto de la computadora.

La tarjeta UP-1 contiene un chip EPM7128SLC84-7. Existe un socket que conecta este chip al cable ByteBlaster. Conecte el cable en ese socket y conecte el otro extremo del cable en el puerto paralelo de la computadora. Asegúrese de que la tarjeta UP-1 está conectada al suministro de corriente y que el "LED de energía" está encendido.

Use File | Open Project para abrir el proyecto *example_vhdl*. Seleccione Tools | Programmer para abrir la ventana del módulo *Programmer* mostrada en la figura D.7. El archivo de programación para el proyecto *example_vhdl*, llamado *example_vhdl.pof,* debe aparecer en la ventana *Programmer*. Si no se muestra haga clic en Edit | Add File y escriba el nombre del archivo.

Figura D.7 Ventana del módulo *Programmer*.

Para especificar que ByteBlaster se va a usar como el hardware de programación, haga clic en el botón Hardware Setup para abrir la ventana de la figura D.8*a*. Si ByteBlaster no aparece en la sección Available hardware items, haga clic en el botón Add Hardware. Con ello se abre la ventana de la figura D.8*b*. Abra la lista desplegable al lado de Hardware type y seleccione el elemento ByteBlasterMV or ByteBlaster II. Haga clic en OK para regresar a la ventana Hardware Setup. Observe que el controlador (*driver*) para el cable ByteBlaster debe instalarse en la computadora que se utilizará para que funcione el procedimiento recién descrito. Si el cable ByteBlaster no aparece en la lista de la figura D.8*b*, entonces ejecute el comando *bblpt/i* desde un indicador de comandos (*prompt*) de Windows. Este comando está disponible en el directorio *C:\quartus\drivers\i386*, siempre que el software Quartus II esté instalado en el directorio *C:\quartus*.



a) La ventana de instalación de hardware (*Hardware Setup*).

b) El cuadro de diálogo para añadir hardware (*Add Hardware*).

Figura D.8 Adición del hardware ByteBlaster.

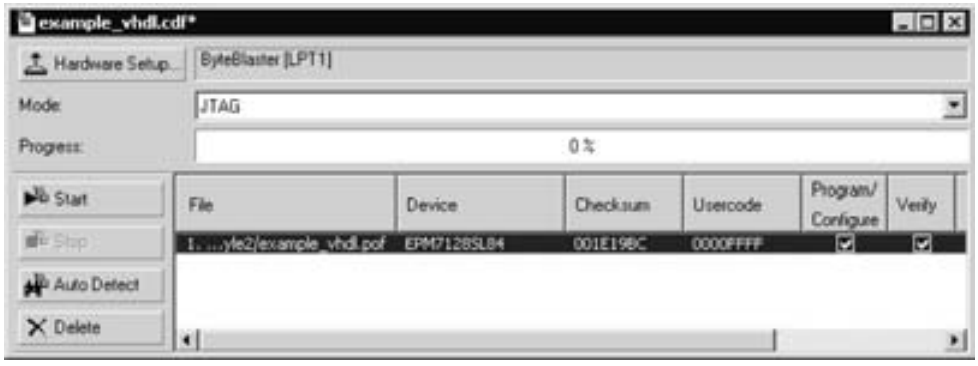


Figura D.9 Ventana final del módulo *Programmer.*

El cable ByteBlaster ahora debe aparecer en la sección Available hardware items. Haga clic en ese elemento para seleccionarlo y luego haga clic en el botón Select Hardware. Cierre la ventana *Hardware Setup* para regresar a la ventana *Programmer* de la figura D.9. Nótese que el ByteBlaster ahora se muestra a la derecha del botón Hardware, lo que significa que este cable ahora está seleccionado. Como se indica en la figura, active con un clic las dos casillas bajo Program/Configure and Verify asociadas con el archivo *example_vhdl.pof.*

Para configurar el chip EPM7128SLC84-7, seleccione Processing | Start Programming. El módulo *Programmer* descarga automáticamente el archivo *example_vhdl.pof* por el cable ByteBlaster hacia el dispositivo y luego verifica que la programación se haya realizado correctamente. El módulo *Programmer* ahora puede cerrarse. El diseñador puede probar el circuito implementado en el chip utilizando el equipo de pruebas apropiado.

La tarjeta UP-1 también contiene un chip FPGA. El procedimiento seguido para descargar un circuito en este chip es semejante al descrito para el dispositivo MAX 7000, pero se precisan algunos pasos más. El lector que intente utilizar el chip FPGA debe remitirse a la documentación que acompaña a la tarjeta UP-1 para hallar instrucciones detalladas.

D.3 COMENTARIOS FINALES

En los tutoriales 1, 2 y 3 expusimos muchas de las funciones más importantes de Quartus II. Sin embargo, hay muchas más. El lector puede aprender acerca de las más avanzadas del sistema CAD explorando los diversos comandos y la ayuda en línea proporcionada en cada aplicación.

apéndice

E

DISPOSITIVOS COMERCIALES

En el capítulo 3 describimos los tres tipos principales de dispositivos lógicos programables (PLD): PLD simples, PLD complejos y arreglos de compuerta programables por campo (FPGA). En este apéndice describimos algunos ejemplos de los productos PLD comerciales.

E.1 PLD SIMPLES

Los PLD simples (SPLD, Simple PLD) comprenden PLA, PAL y otros tipos de dispositivos similares. Los principales fabricantes de productos PLD se enumeran en la tabla E.1. En la primera y la segunda columnas se muestra el nombre de la compañía y algunos de los productos SPLD que ofrecen. Las hojas de datos que describen cada producto pueden obtenerse en la World Wide Web (WWW), con la dirección indicada en la tercera columna de la tabla.

Tabla E.1 Productos SPLD comerciales.

Fabricante	Producto SPLD	Dirección en la WWW
Altera	Classic	http://www.altera.com
Atmel	PAL	http://www.atmel.com
Cypress	PAL	http://www.cypress.com
Lattice	ispGAL	http://www.latticesemi.com

E.1.1 EL DISPOSITIVO PAL 22V10

Los dispositivos PAL se cuentan entre los SPLD más comunes. Se ofrecen en diversos tamaños y se identifican por medio de un número de parte con la forma *NNXMM–S*. Los dígitos *NN* especifican el número total de pines de entrada y salida; los dígitos *MM* dan el número de pines que pueden utilizarse como salidas. La letra *X* brinda información adicional, por ejemplo, si el PAL

contiene flip-flops. El dígito final, *S,* especifica el *grado de velocidad.* El valor representa el retraso de propagación desde un pin de entrada en el PAL hasta un pin de salida, suponiendo que se evita el flip-flop, si está presente.

Un ejemplo de un PAL usado comúnmente es el 22V10 [1], que aparece en la figura E.1. Hay 11 pines de entrada que alimentan el plano AND y una entrada adicional que también puede servir como

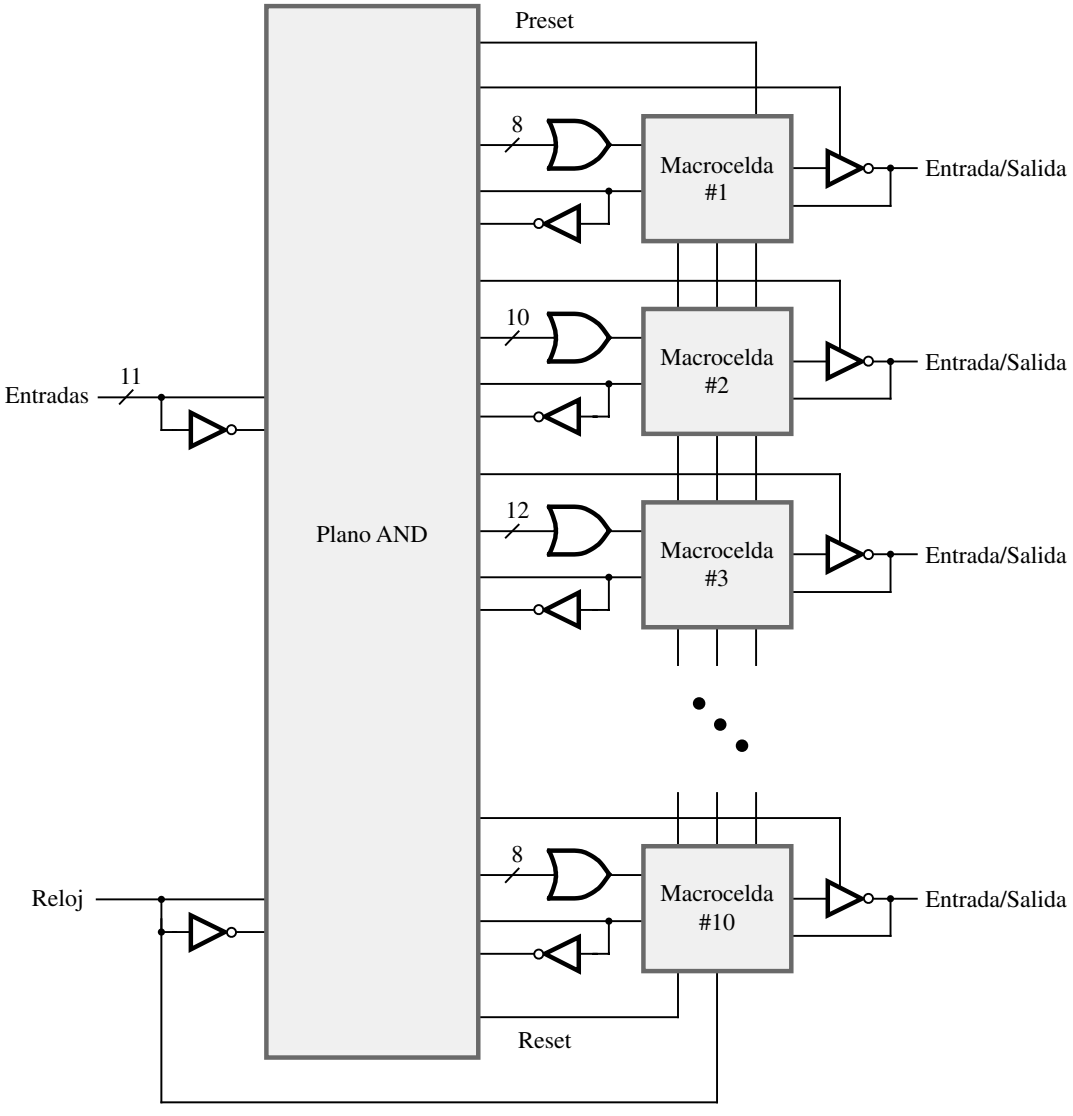


Figura E.1 El dispositivo PAL 22V10.

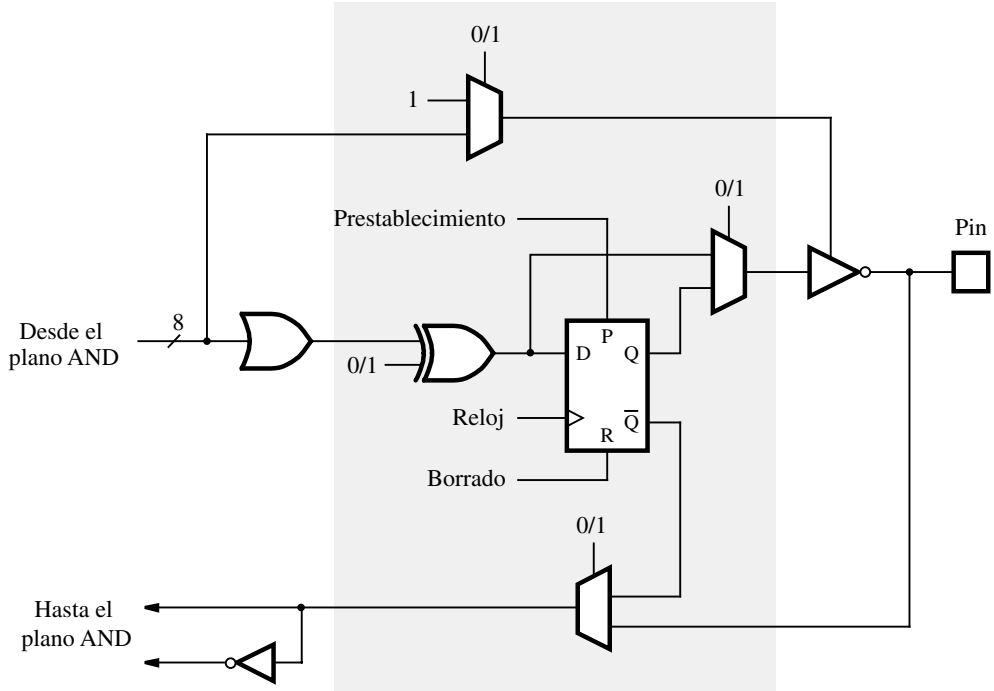


Figura E.2 La macrocelda 22V10.

una entrada de reloj. Las compuertas OR varían en tamaño desde 8 hasta 16 entradas. Cada pin de salida tiene un buffer triestado que permite al pin utilizarse de manera optativa como entrada.

En la sección 3.6.2 dijimos que el sistema de circuitos entre una compuerta OR y una salida de un PAL se llama *macrocelda.* En la figura E.2 se muestra una de las macroceldas del PAL 22V10, la cual conecta la compuerta OR mostrada a una entrada de una compuerta XOR, que alimenta un flip-flop D. Como la otra entrada a la compuerta XOR puede programarse para ser 0 o 1, es posible usarla para complementar la salida de la compuerta OR. Un multiplexor dos a uno permite evitar el flip-flop y el buffer triestado puede habilitarse en forma permanente o conectarse al término producto desde el plano AND. O bien, la salida $\overline{Q}$ desde el flip-flop o la salida del buffer triestado pueden conectarse al plano AND. Si el buffer está inhabilitado, el pin correspondiente puede utilizarse como una entrada.

E.2 PLD COMPLEJOS

Los nombres de varios fabricantes de PLD complejos (CPLD, Complex PLD), los productos que ofrecen y las direcciones web correspondientes se presentan en la tabla E.2. Un ejemplo de una familia CPLD muy utilizada, Altera MAX 7000 [2], se describe en la sección siguiente.

Tabla E.2 Productos CPLD comerciales.

Fabricante	Producto CPLD	Dirección en la WWW
Altera	MAX 3000, 7000 y 9000, y MAX II	http://www.altera.com
Atmel	ATF	http://www.atmel.com
Cypress	Delta39K, FLASH370, Ultra37000	http://www.cypress.com
Lattice	ispLSI, ispMACH	http://www.latticesemi.com
Xilinx	XC9500, CoolRunner	http://www.xilinx.com

E.2.1 MAX 7000 DE ALTERA

La familia de CPLD MAX 7000 incluye chips de diferentes tamaños, desde el 7032, que tiene 32 macroceldas, hasta el 7512, con 512. Hay dos variantes principales de estos chips, identificadas por el sufijo *S:* si lo lleva el nombre del chip, como en 7128S, entonces el chip es programable dentro del sistema, pero si no es así, como en 7128, entonces el chip debe programarse en una unidad de programación.

La estructura general de un chip MAX 7000 se ilustra en la figura E.3. Hay cuatro pines de entrada dedicados; dos de ellos sirven como entradas de reloj globales y uno puede usarse como un reset global para todos los flip-flops. Cada cuadro sombreado en la figura se llama *bloque*

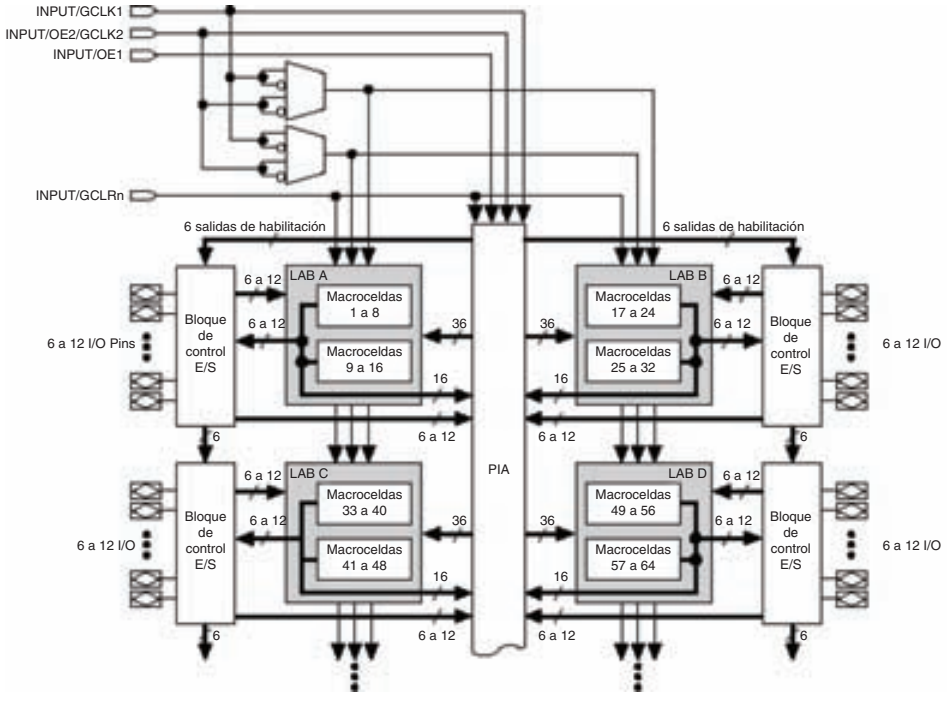


Figura E.3 CPLD MAX 7000 (cortesía de Altera).

de arreglo lógico (LAB logic array block) y contiene 16 macroceldas. Cada LAB está conectado a un *bloque de control de E/S (I/O),* el cual contiene buffers triestado conectados a pines del encapsulado; cada uno de estos pines puede emplearse como un pin de entrada o de salida. Cada LAB también está conectado al *arreglo de interconexión programable (PIA, programmable interconnect array).* El PIA se compone de un conjunto de cables que abarcan el dispositivo entero. Todas las conexiones entre macroceldas se realizan utilizando el PIA.

En la figura E.4 se muestra la estructura de una macrocelda MAX 7000. Hay cinco términos producto que pueden conectarse a través de la *matriz de selección de términos producto* a una compuerta OR, la cual puede configurarse para usar sólo los términos producto necesarios para la función lógica que está implementándose en la macrocelda. Si se requieren más de cinco términos producto pueden "compartirse" desde otras macroceldas, como se describe enseguida. La compuerta OR está conectada por medio de una compuerta XOR a un flip-flop, el cual puede evitarse.

En la figura E.5 se muestra cómo pueden compartirse los términos producto entre las macroceldas. La compuerta OR de una macrocelda incluye una entrada adicional que puede conectarse a la salida de la compuerta OR en la macrocelda encima de ella. Esta característica se llama *expansores paralelos* y se usa para funciones lógicas de hasta 20 términos producto. Si se precisan más, entonces se utiliza una característica conocida como *expansores compartidos*. Como se muestra en el cuadro sombreado inferior de la figura E.4, uno de los términos producto de una macrocelda se invierte y es realimentado al arreglo de términos producto. Si las entradas a estos términos se emplean en su forma complementada, entonces la aplicación del teorema de DeMorgan produce un término suma. Un expansor compartido puede ser utilizado por cualquier macrocelda en el mismo LAB.

Cada dispositivo MAX 7000 específico está disponible en diversos grados de velocidad, los cuales indican el retraso de propagación desde un pin de entrada a través del PIA y de una macrocelda a un pin de salida. Por ejemplo, el chip llamado 7128S-7 tiene un retraso de propagación de 7.5 ns. Si la función lógica implementada usa expansores paralelos o compartidos, ese retraso aumenta.

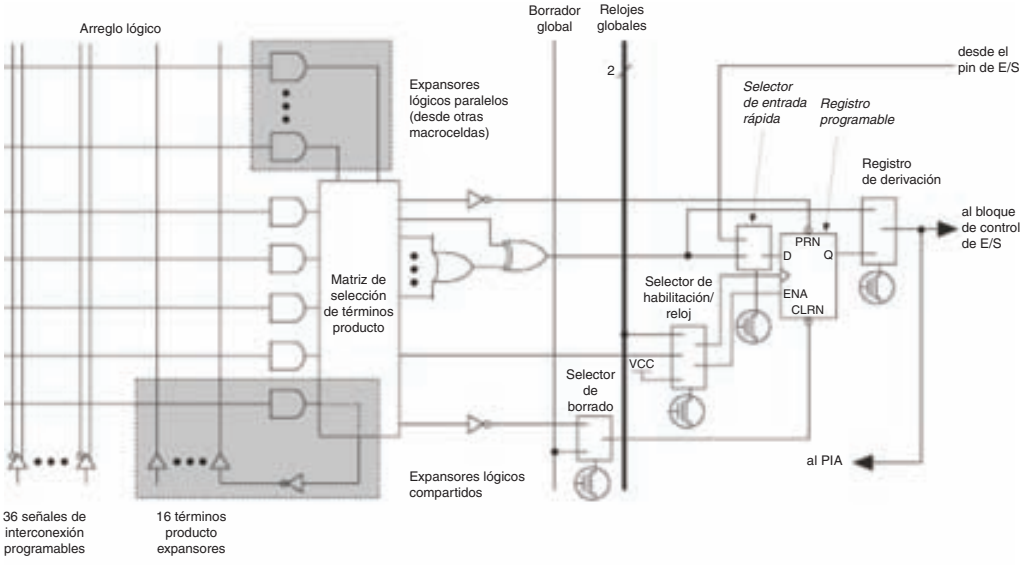


Figura E.4 Macrocelda MAX 7000 (cortesía de Altera).

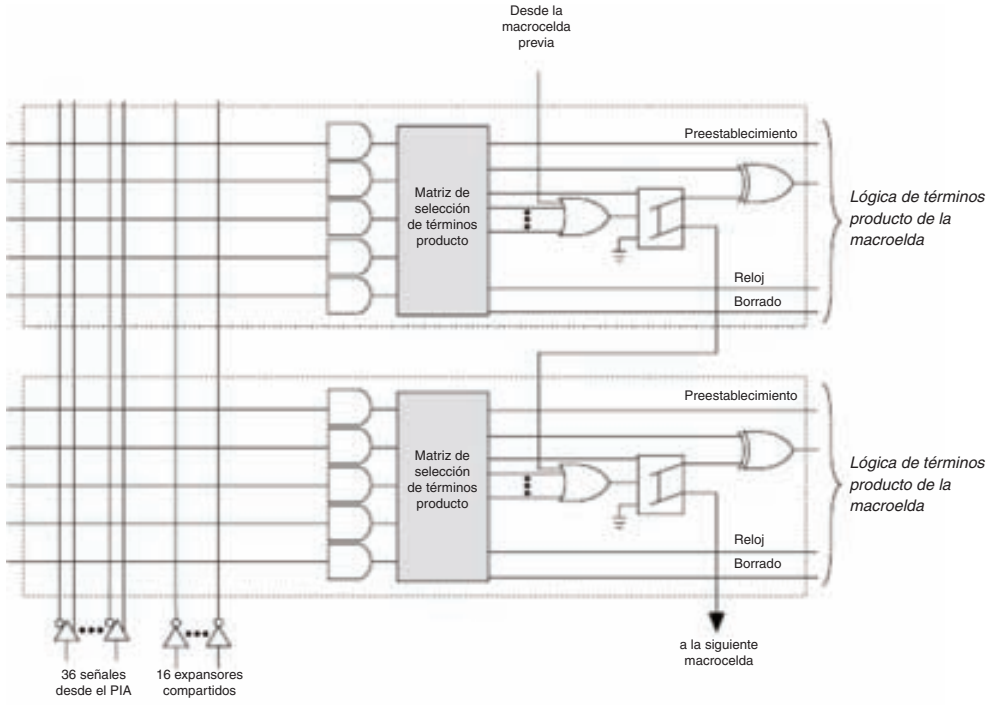


Figura E.5 Expansores paralelos (cortesía de Altera).

E.3 ARREGLOS DE COMPUERTA PROGRAMABLES POR CAMPO

En la tabla E.3 se listan los nombres de los fabricantes de FPGA, sus productos y sus direcciones web. En esta sección describimos ejemplos de FPGA producidos por Altera y Xilinx.

E.3.1 FLEX 10K DE ALTERA

En la figura E.6 se muestra la estructura del chip FLEX 10K [3]. Contiene un conjunto de bloques de arreglo lógico (LAB), cada uno de los cuales comprende ocho *elementos lógicos* basado en tablas de búsqueda (LUT, *lookup table*). Además de los LAB, el chip también contiene bloques de arreglos embebidos (EAB, *embedded array blocks*), que son bloques de SRAM que pueden configurarse para proporcionar bloques de memoria de distintas proporciones (véase la sección 10.1.3). Los LAB y los EAB pueden interconectarse mediante *cables de interconexión* para filas y columnas. Estos cables también proporcionan conexiones a los pines de entrada y salida en el encapsulado.

En la figura E.7 se presenta el contenido de un LAB. Tiene una serie de entradas que son provistas desde los cables de interconexión de la fila adyacente hasta un conjunto de cables de interconexión locales dentro del LAB. Estos cables locales sirven para hacer conexiones a las entradas de los elementos lógicos, y las salidas de los elementos lógicos también retroalimentan los cables locales. Las salidas de los elementos lógicos también se conectan a los cables de columna y de fila adyacentes. La estructura del elemento lógico se representa en la figura E.8. El elemento

Tabla E.3 Productos FPGA comerciales.

Fabricante	Productos FPGA	Dirección en la WWW
Actel	Act 1, 2 y 3, MX, SX	http://www.actel.com
Altera	FLEX 6000, 8000 y 10K, Mercury, APEX 20K (II), Excalibur, Stratix (II)	http://www.altera.com
Atmel	AT6000, AT40K	http://www.atmel.com
Lattice	ispXPGA, ORCA	http://www.latticesemi.com
QuickLogic	pASIC, Eclipse, Eclipse II	http://www.quicklogic.com
Xilinx	XC3000, XC4000, Spartan (3), Virtex, Virtex II (Pro)	http://www.xilinx.com

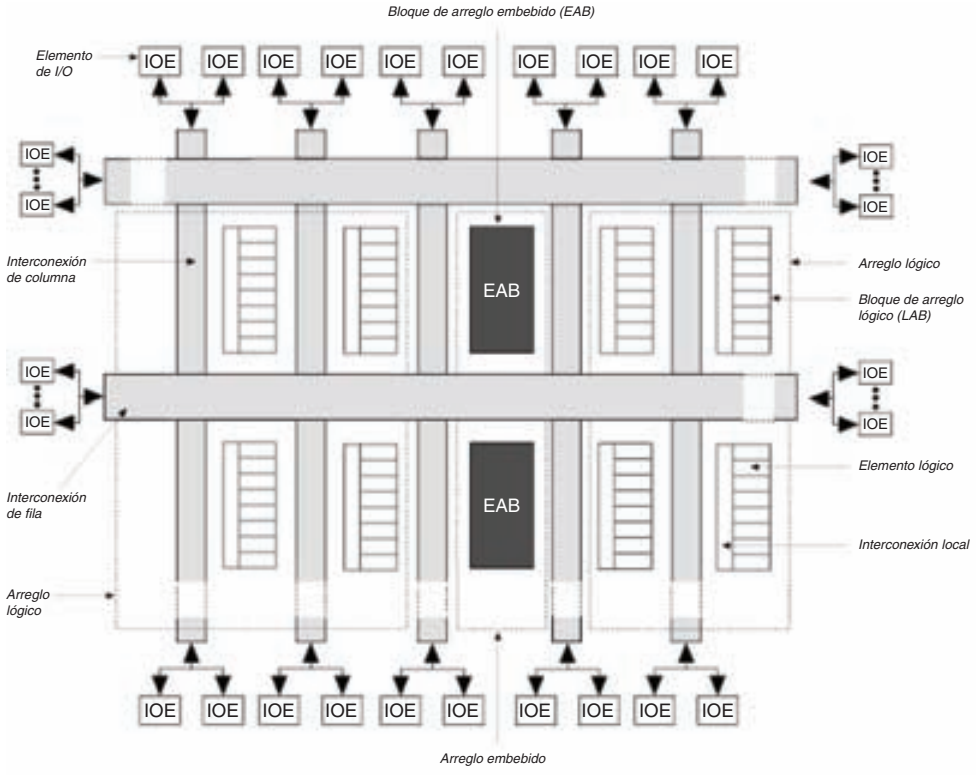


Figura E.6 FPGA FLEX 10K (cortesía de Altera).

tiene una LUT de cuatro entradas y un flip-flop que puede evitarse. Para la implementación de los sumadores aritméticos, la LUT de cuatro entradas puede utilizarse para implementar dos funciones de tres entradas: las funciones de suma y acarreo de un sumador completo.

En la figura E.9 se describe la estructura de un EAB. Contiene 2048 celdas de SRAM, que se utilizan para proporcionar bloques de memoria que presentan una variedad de proporciones: 256×8, 512×4, 1024×2 y 2048×1 bits. Las entradas de dirección y de datos al bloque

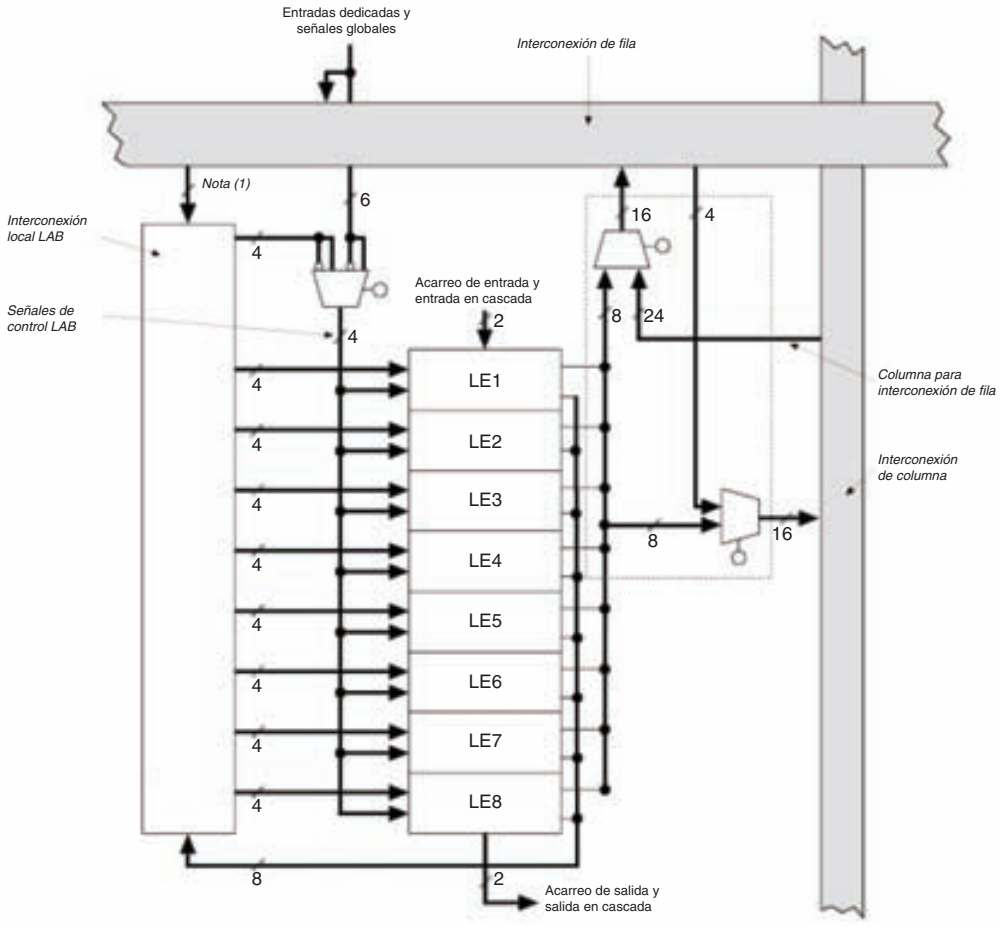


Figura E.7 Bloque de arreglo lógico FLEX 10K (cortesía de Altera).

de memoria son provistas desde un conjunto de cables de interconexión locales. Esas entradas, igual que un enable de escritura para el bloque de memoria, pueden almacenarse de manera optativa en los flip-flops. En la figura E.9 se indica que el número de direcciones y entradas de datos conectadas al bloque de memoria varía según la razón de aspecto que se use. Asimismo, las salidas de datos también pueden guardarse opcionalmente en los flip-flops. Para los bloques de memoria grandes es posible combinar varios EAB.

La configuración de los EAB se realiza mediante módulos prediseñados, semejantes a los de la biblioteca LPM. Por ejemplo, el módulo llamado *lpm_ram_dq* sirve para especificar un bloque de SRAM, y *lpm_rom,* para un bloque ROM. Esos módulos pueden importarse a un esquema o instanciarse en código por medio de un lenguaje como VHDL. Al programar el chip FPGA es posible indicar los datos iniciales que se cargarán en el bloque de memoria. Ello se lleva a cabo creando un tipo de archivo especial, denominado *archivo de inicialización de memoria,* que se asocia con el módulo *lpm_ram_dq* o *lpm_rom*. En la documentación de MAX+plusII se hallarán más detalles acerca del uso de tales módulos.

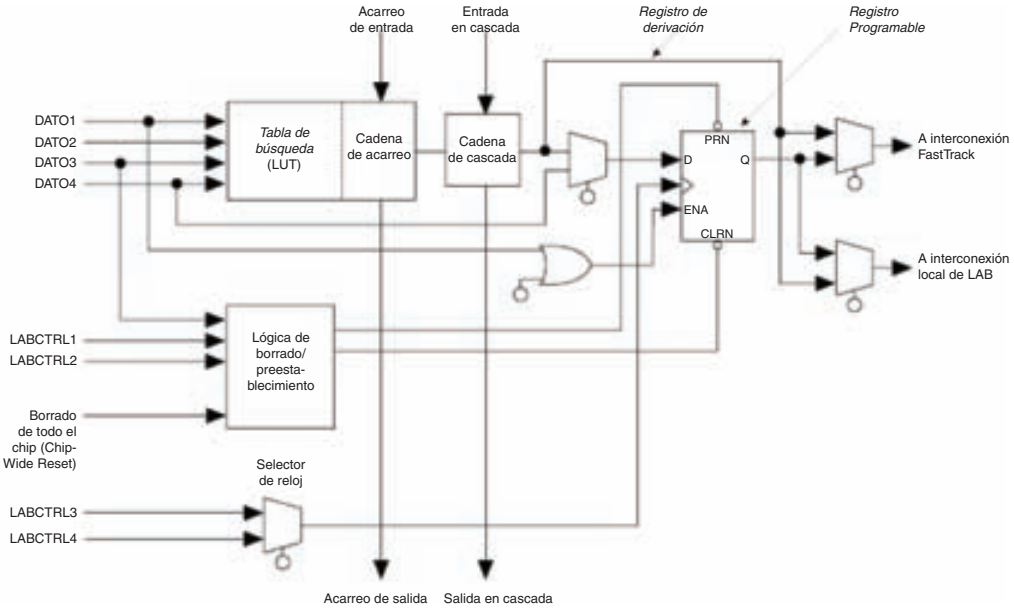


Figura E.8 Elemento lógico FLEX 10K (cortesía de Altera).

Los chips FLEX 10K están disponibles en tamaños que van desde 10K10 hasta 10K250, lo que ofrece alrededor de 10,000 y 250,000 compuertas lógicas equivalentes, respectivamente. Los chips específicos están disponibles en diversas velocidades, que se indican usando una letra como sufijo, por ejemplo *A*, como en 10K10A, y un grado de velocidad, como en 10K10A-1. A diferencia de los PAL y CPLD, el grado de velocidad de un FPGA no especifica un retraso de propagación real en nanosegundos, sino una velocidad relativa dentro de la familia de dispositivos. Por ejemplo, el chip 10K10-1 es más rápido que el chip 10K10-2. Los retraso de propagación reales en los circuitos implementados pueden examinarse utilizando una herramienta CAD simuladora de tiempo.

E.3.2 XC4000 DE XILINX

La estructura de un chip XC4000 de Xilinx [4] es similar a la del FPGA mostrada en la figura 3.35. Tiene un arreglo bidimensional de *bloques lógicos configurables* (CLB, *configurable logic blocks*) que pueden interconectarse usando los canales de enrutamiento verticales y horizontales. Los chips varían en tamaño del XC4002 al XC40250, que tienen alrededor de 2000 y 250,000 compuertas lógicas equivalentes, respectivamente. Como se muestra en la figura E.10, un CLB contiene dos LUT de cuatro entradas; por consiguiente, puede implementar cualesquiera dos funciones lógicas de cuatro variables. La salida de cada una de estas LUT puede almacenarse de manera opcional en un flip-flop. El CLB también contiene una LUT de tres entradas conectadas a las dos LUT de cuatro entradas, lo que permite la implementación de funciones con cinco o más variables.

Similar a los elementos lógicos de los FPGA FLEX 10K descritos en la sección E.3.1, el CLB puede configurarse para una implementación eficaz de los módulos sumadores. En este modo

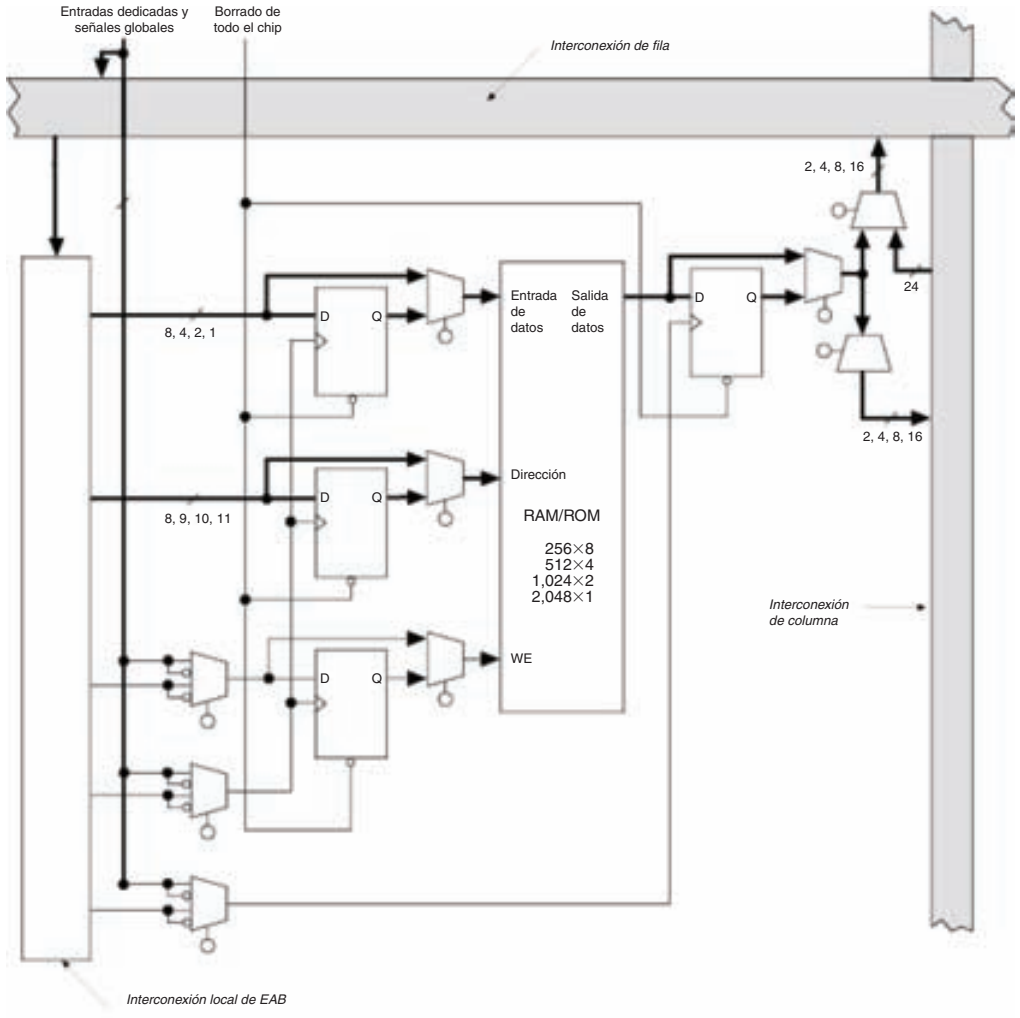


Figura E.9 Bloque de arreglo embebido (cortesía de Altera).

cada LUT de cuatro entradas en el CLB implementa tanto las funciones de suma como de acarreo de un sumador completo. Además, en vez de implementar las funciones lógicas, el CLB puede usarse como un módulo de memoria. Cada LUT de cuatro entradas puede servir como un bloque de memoria de 16 × 1, o las dos LUT de cuatro entradas pueden combinarse en un bloque de memoria de 32 × 1. Varios CLB pueden combinarse para formar bloques de memoria más grandes.

Los CLB se interconectan usando los cables de los canales de enrutamiento. Se proporcionan cables de varias longitudes, desde los que abarcan un solo CLB hasta los que cubren todo el dispositivo. El número de cables en un canal de enrutamiento varía para cada chip específico.

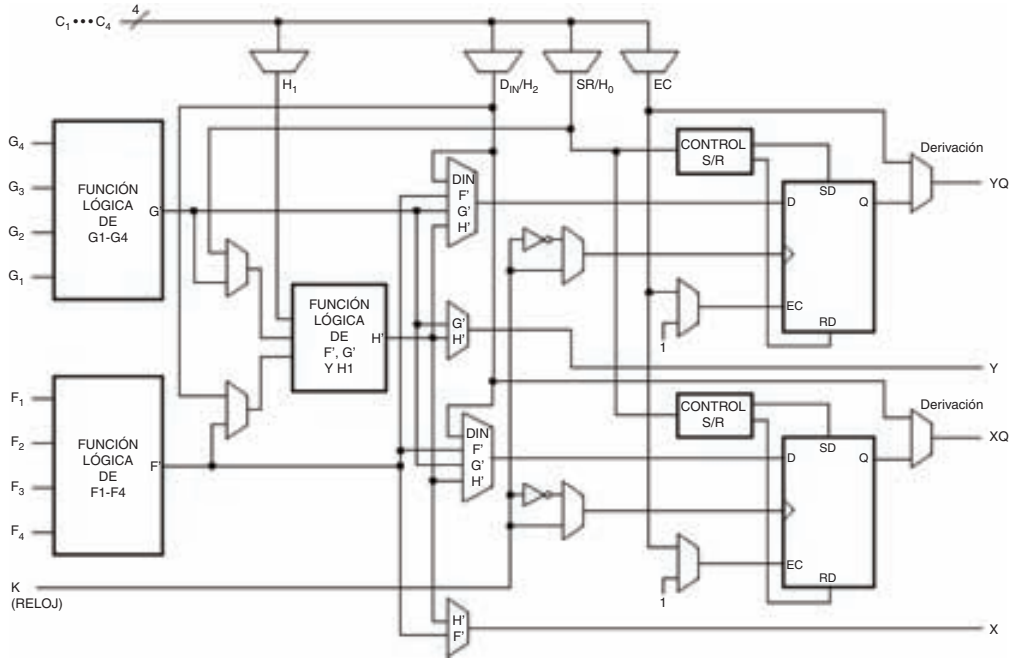


Figura E.10 Bloque lógico configurable XC4000 (cortesía de Xilinx).

E.3.3 APEX 20K de Altera

La familia APEX 20K [5] de Altera es la siguiente generación de productos después del FLEX 10K. El elemento lógico (LE, *logic element*), que es una versión mejorada del representado en la figura E.8, contiene una LUT de cuatro entradas y un flip-flop. Los chips varían en tamaño de 1200 a 51, 840 LE.

Cada dispositivo APEX contiene elementos lógicos (LUT), bloques de memoria y celdas IO. Los LE están arreglados en LAB, parecidos a la estructura que se muestra en la figura E.7, con 10 LE por LAB. Los LAB se agrupan después en MegaLAB, con hasta 24 LAB por MegaLAB. Como se muestra en la figura E.11, el MegaLAB contiene cables para interconectar los LAB y también un bloque de memoria, llamado *bloque de sistema embebido* (ESB, *embedded system block*). El ESB, similar al EAB mostrado en la figura E.9, soporta los bloques de memoria con varias proporciones. Un dispositivo APEX se compone de dos o cuatro columnas de MegaLAB; el número de MegaLAB por columna varía en cada dispositivo.

E.3.4 Stratix de Altera

Stratix [6] es el producto FPGA de Altera que remplaza a la familia APEX. En la figura E.12 se muestra la arquitectura de un dispositivo Stratix. Cada chip se compone de columnas de recursos de varios tipos. Las columnas LAB albergan elementos lógicos arreglados en LAB que tienen 10 LE por LAB. Cada LE contiene una LUT de cuatro entradas y un registro, y puede configurarse

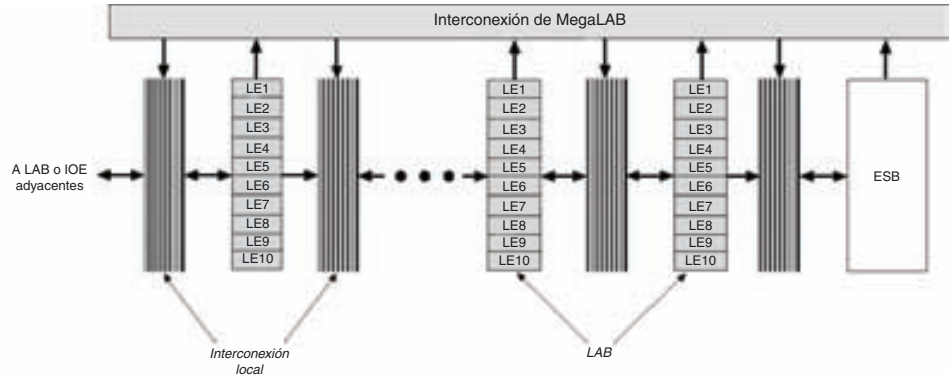


Figura E.11 MegaLAB APEX 20K (cortesía de Altera).

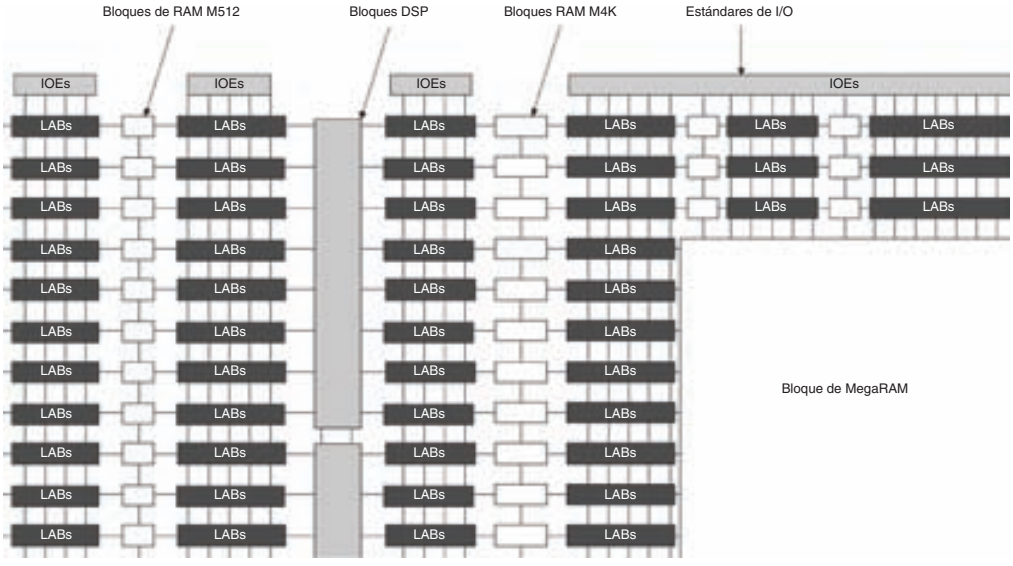


Figura E.12 LAB, DSP y bloques de memoria Stratix (cortesía de Altera).

en una variedad de modos, incluido un modo aritmético rápido. Hay una serie de tipos de recursos de alambrado en un chip Stratix. Las conexiones dentro de un LAB se hacen utilizando recursos locales rápidos, como una cadena de acarreo que se ejecuta hacia abajo en cada columna. Para las conexiones de un LAB a otros recursos existen conexiones cortas al vecino más cercano, cables que abarcan cuatro columnas o filas y aun más largos.

Además de las columnas LAB, los dispositivos Stratix contienen otros tres tipos de columnas. Las columnas M512 se componen de bloques de memoria con 512 bits cada uno, y las columnas M4K contienen bloques de memoria más grandes con 4K bits por bloque. Cada uno de los bloques M512 y M4K soporta implementaciones de memorias con varias proporciones. Los dispositivos Stratix también incluyen bloques de memoria muy grandes, llamados MegaRAM, cada uno de los cuales contienen 512K bits de memoria.

Finalmente, hay dos columnas que comprenden los bloques de procesamiento de señales digitales (DSP, *Digital Signal Processing*). Cada uno de estos bloques incluye circuitos de hardware multiplicador y sumador que permiten una rápida multiplicación y acumulación (suma) de los datos. Estos bloques proporcionan una implementación eficaz de los tipos de circuitos empleados en las aplicaciones de procesamiento de señales digitales.

Los chips Stratix están disponibles en tamaños que van de 10 570 a 79 040 elementos lógicos y hasta siete Mbits de memoria.

E.3.5 Cyclone de Altera

Los FPGA Cyclone [7] se basan en la arquitectura Stratix, pero están dirigidos a aplicaciones de bajo costo. Un chip Cyclone tiene la misma estructura básica mostrada en la figura E.12, excepto que las columnas DSP se eliminaron y sólo se incluyen bloques de memoria de M4K. El elemento lógico de un chip Cyclone es una LUT de cuatro entradas con un sistema de circuitos aritméticos dedicado y un flip-flop programable. Los dispositivos Cyclone varían en tamaño de 2 910 a 20 060 elementos lógicos y 288 Kbits de memoria.

E.3.6 Stratix II de Altera

Los FPGA Stratix II [8] son el sucesor de la familia Stratix. Ofrecen tamaños de dispositivos de 15 600 a 179 400 elementos lógicos y hasta nueve Mbits de memoria. Stratix II contiene un elemento lógico más complejo que otros FPGA, llamado *módulo lógico adaptativo* (ALM, *Adaptive Logic Module*). Como se muestra en la figura E.13, el ALM se compone de un circuito lógico combinacional y dos flip-flops programables. El circuito lógico combinacional puede programarse como una o dos LUT; puede implementar una sola función lógica de hasta siete

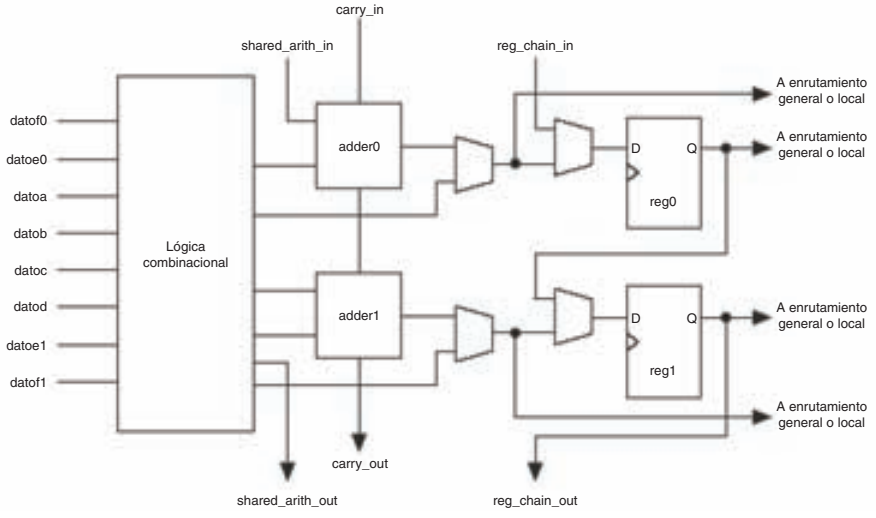


Figura E.13 El módulo lógico adaptativo Stratix II.

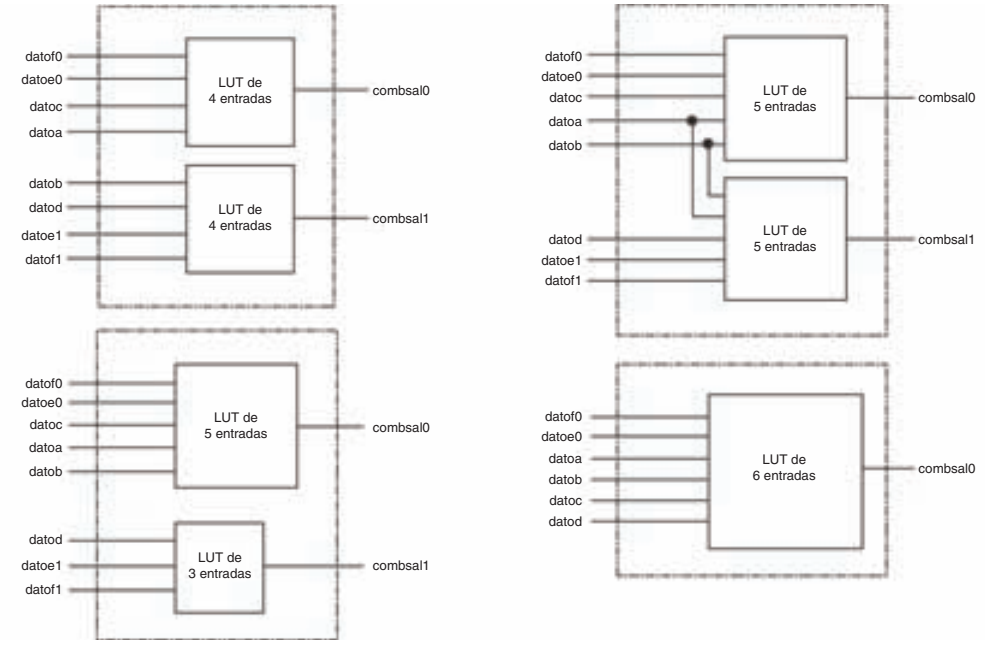


Figura E.14 Algunos de los modos del ALM de los FPGA Stratix II.

entradas, o dos funciones de varios tamaños. En la figura E.14 se muestran algunas de las combinaciones posibles del ALM, como la producción de dos LUT de cuatro entradas, una LUT de cuatro entradas más una de cinco, etcétera.

E.3.7 VIRTEX DE XILINX

Los FPGA Virtex de Xilinx [9] son la familia de la siguiente generación después del XC4000. Como se indica en la figura E.15, cada chip Virtex se compone de recursos lógicos, llamados CLB, y recursos de memoria, denominados RAM de bloque (BRAM, *Block RAM*). El CLB es una versión mejorada del CLB XC4000 CLB mostrado en la figura E.10. Como se señala en la figura E.16, el CLB de Virtex se divide en dos; cada mitad se llama *pieza.* Cada pieza contiene dos LUT de cuatro entradas, dos registros y lógica de aritmética dedicada (cadena de acarreo).

Los bloques BRAM contienen 4K bits de memoria y pueden configurarse para soportar razones de aspecto específicas de 4096 × 1 a 256 × 16. Los bloques CLB y BRAM pueden interconectarse por medio de cables que abarcan un solo CLB o distancias más grandes. Los dispositivos Virtex se venden en tamaños de 256 a 46 592 piezas de CLB.

E.3.8 VIRTEX-II Y VIRTEX-II PRO DE XILINX

Los FPGA Virtex-II [10] y Virtex-II Pro [11] de Xilinx son los sucesores de la familia Virtex. Se ofrecen en tamaños de 3168 a 99,216 elementos lógicos y con más de ocho Mbits de memoria. Los elementos lógicos se acomodan en piezas similares a los FPGA de Virtex (véase la figura E.16),

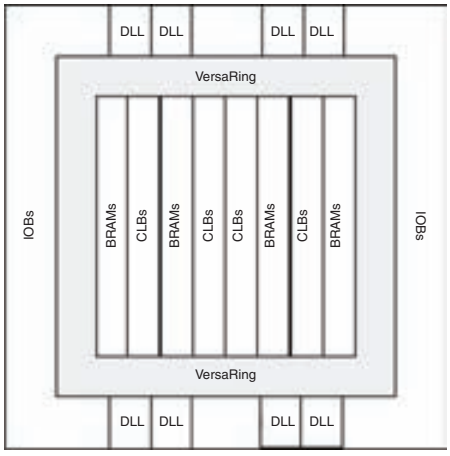


Figura E.15 FPGA de Virtex (cortesía de Xilinx).

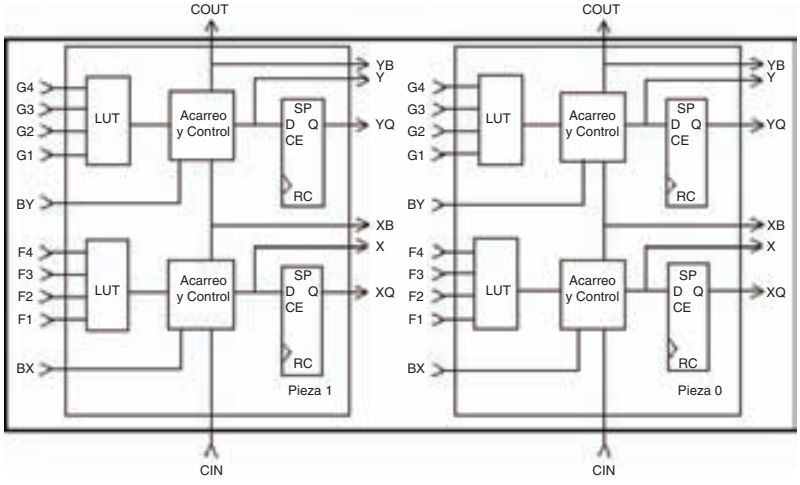


Figura E.16 Bloque lógico de Virtex (cortesía de Xilinx).

con cuatro piezas en un CLB. Los chips Virtex-II Pro incluyen uno o más núcleos de microprocesador dentro del chip, y tienen funciones avanzadas adicionales que no están presentes en Virtex-II.

E.3.9 Spartan-3 de Xilinx

Los FPGA Spartan-3 de Xilinx [12] son una versión de bajo costo de la arquitectura Virtex-II. Los elementos lógicos, semejantes a los de Virtex-II, se arreglan en CLB, y cada uno de estos blo-

ques tiene cuatro piezas, pero no todas ellas poseen el mismo conjunto de funciones que en Virtex-II. Los chips Spartan-3 están disponibles en tamaños de 1728 a 74,880 elementos lógicos y más de 1.8 Mbits de memoria.

E.4 LÓGICA DE TRANSISTOR A TRANSISTOR

Antes de la aparición de CMOS, la tecnología dominante era la *lógica de transistor a transistor,* comúnmente conocida como *TTL.* La mayor parte de los sistemas digitales construidos en las décadas de 1970 y 1980 se basó en esta tecnología. Los circuitos TTL están disponibles en tamaños relativamente pequeños, conocidos como integración de pequeña escala (SSI, *small-scale integration*) e integración de mediana escala (MSI, *medium-scale integration*), como se explicó en la sección 3.5. Un chip SSI típico contiene sólo unas cuantas compuertas lógicas, con sus entradas y salidas disponibles en los pines del encapsulado. Un chip MSI puede comprender un circuito un tanto más grande, como una unidad lógica y aritmética (ALU) de cuatro bits.

La tecnología TTL no es tan adecuada para la integración en gran escala como la tecnología CMOS, la cual ha conducido a la desaparición del TTL. Sin embargo, su impacto fue tan grande que algunos aspectos aún son importantes en la actualidad. En esta sección consideramos esos aspectos.

Niveles de voltaje

Los circuitos TTL utilizan un suministro de corriente de 5 voltios. Cualquier voltaje en los límites de 0 a 0.8 V se interpreta como un 0 lógico cuando se aplica a un pin de entrada. Un voltaje en los límites de 2 a 5 se interpreta como un 1 lógico. Utilizando la terminología de la sección 3.8, $V_{IL} = 0.8$ V y $V_{IH} = 2$ V. El voltaje de salida máximo producido para el 0 lógico es $V_{OL} = 0.4$ V, y el voltaje mínimo producido para el 1 lógico es $V_{OH} = 2.4$ V. Estos parámetros llevan a los márgenes de ruido $NM_L = NM_H = 0.4$ V. Los voltajes de salida típicos generados por un circuito TTL son 0.2 V para el 0 y 3.6 V para el 1 lógico.

Cuando se diseña un circuito digital nuevo, con frecuencia su empleo está dirigido a un sistema digital existente. Si se usan diferentes tecnologías para implementar distintas partes de un sistema, es esencial asegurar que los niveles de voltaje compatibles se utilizan para señales en las interfaces específicas entre las distintas partes. Aun cuando los niveles de voltaje de CMOS normalmente son diferentes de los niveles de TTL, algunos chips CMOS, como los PLD, pueden configurarse para usar los niveles de voltaje compatibles con TTL en sus pines de entrada y salida.

Conexiones de entrada

En los circuitos CMOS todas las entradas a una compuerta siempre deben manejarse para cualquier valor lógico 0 o 1. De lo contrario, la salida de la compuerta tendrá un valor desconocido (por lo general, triestado). En el caso de los circuitos TTL, una entrada sin conectar se comporta como si estuviera conectada a una constante 1.

E.4.1 FAMILIAS DE CIRCUITOS TTL

Los circuitos TTL están disponibles en varios diseños que tienen diferentes velocidades de propagación y consumo de energía. Tienen las mismas características funcionales, definidas por las especificaciones del tipo de circuitos conocido como la serie 7400, la cual se presentó en la sección 3.5. En realidad, la etiqueta 7400 indica un chip que se compone de cuatro compuertas NAND de dos entradas. Otros chips que contienen elementos lógicos diferentes tienen el mismo

prefijo 74, pero se identifican por medio de dígitos adicionales. Por ejemplo, 7421 indica un chip que se compone de dos compuertas AND de cuatro entradas. En la tabla E.4 se presentan las características de retraso de propagación y disipación de energía de las diversas familias TTL.

Tabla E.4 Familias lógicas TTL.

Familias	Designación	Retraso de propagación (en ns)	Disipación de energía (en mW)
Standard	7400	9	10
Low power	74L00	33	1
High speed	74H00	6	22
Schottky	74S00	3	20
Low-power Schottky	74LS00	9	2
Advanced Schottky	74AS00	1.5	20
Advanced low-power Schottky	74ALS00	4	1
Fast	74F00	3	4

El TTL estándar (Standard) se basa en las especificaciones originales y fue el primer tipo de estos circuitos introducido en la década de 1960. Las versiones siguientes proporcionaron varias mejoras. Se desarrollaron circuitos más rápidos, compensando el mayor consumo de energía por retrasos de propagación más cortos. A la inversa, se desarrollaron circuitos de bajo consumo de energía al costo de retrasos de propagación más largos. En la tabla E.4 se presentan los valores típicos que cabe esperar en condiciones de operación normales.

La carga de salida máxima de los circuitos TTL es 10 en el grueso de los casos, pero puede llegar a 20 para los tipos que consumen poca energía. La carga de entrada está determinada por el número de entradas provista en un chip específico.

Las compuertas TTL pueden tener diferentes configuraciones de entrada. Además de la configuración de salida normal, existen compuertas que tienen salidas triestado o salidas de colector abierto. El propósito de una salida triestado se estudia en la sección 3.8.8. Las compuertas con salidas de colector abierto se usan cuando se quiere conectar las salidas de dos o más compuertas en conjunto directamente. Estas compuertas no son dañadas por una conexión como ésta, ya que cada compuerta maneja la salida a 0 o bien no la afecta en lo más mínimo. Al conectar las salidas de varias compuertas de colector abierto a través de una resistencia a $+5$ V se produce un circuito donde el voltaje en el punto de salida es igual a $+5$ V si ninguna de las compuertas produce una salida de 0, y es igual a 0 si una o más compuertas producen la salida de 0. Un método parecido puede usarse con la tecnología CMOS, lo cual da como resultado compuertas de drenaje abierto.

No hemos estudiado la tecnología TTL con detalle porque su importancia ha disminuido en el entorno de diseño actual. Un lector interesado puede consultar varios libros que brindan una explicación detallada. Una referencia particularmente meticulosa es [13].

BIBLIOGRAFÍA

1. Lattice Semiconductor, Simple PLDs Data Sheets, http://www.latticesemi.com
2. Altera Corporation, MAX 7000 CPLD Data Sheets, http://www.altera.com
3. Altera Corporation, FLEX 10K Data Sheets, http://www.altera.com
4. Xilinx Corporation, XC4000 FPGA Data Sheets, http://www.xilinx.com
5. Altera Corporation, APEX 20K Data Sheets, http://www.altera.com
6. Altera Corporation, Stratix FPGA Data Sheets, http://www.altera.com
7. Altera Corporation, Cyclone FPGA Data Sheets, http://www.altera.com
8. Altera Corporation, Stratix II FPGA Data Sheets, http://www.altera.com
9. Xilinx Corporation, Virtex FPGA Data Sheets, http://www.xilinx.com
10. Xilinx Corporation, Virtex-II FPGA Data Sheets, http://www.xilinx.com
11. Xilinx Corporation, Virtex-II Pro FPGA Data Sheets, http://www.xilinx.com
12. Xilinx Corporation, Spartan-3 FPGA Data Sheets, http://www.xilinx.com
13. A. S. Sedra y K. C. Smith, *Microelectronic Circuits,* 5a. ed. (Oxford University Press, Nueva York, 2003).

Capítulo 2

2.7. *a*) Sí *b*) Sí *c*) No

2.12. $f = x_1x_3 + x_2x_3 + \bar{x}_2\bar{x}_3$

2.15. $f = (x_1 + x_2)(\bar{x}_2 + x_3)$

2.20. $f = x_2x_3 + x_1\bar{x}_3$

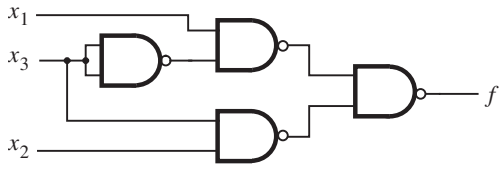
2.23. $f = (x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$

2.28. $f = x_1x_2 + x_1x_3 + x_2x_3$

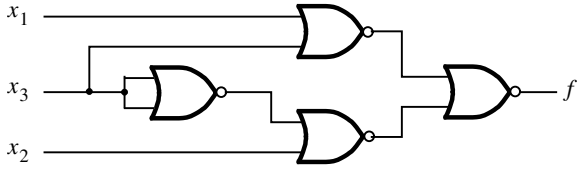
2.32. $f = (x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2 + x_3)$

2.33. $f = \bar{x}_1x_3 + \bar{x}_1x_2 + x_2x_3 + x_1\,\bar{x}_2\,\bar{x}_3$

2.40. El circuito es

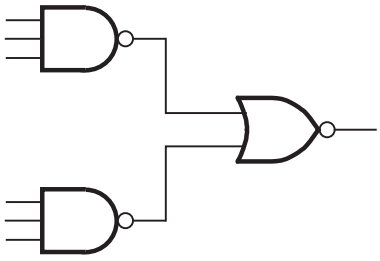


2.42. El circuito es



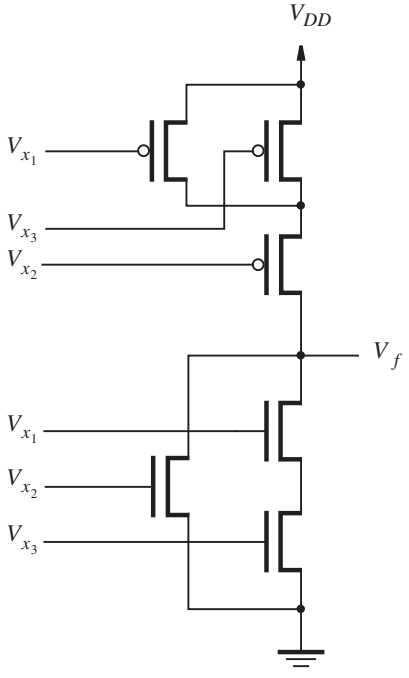
CAPÍTULO 3

3.4. Usando el circuito

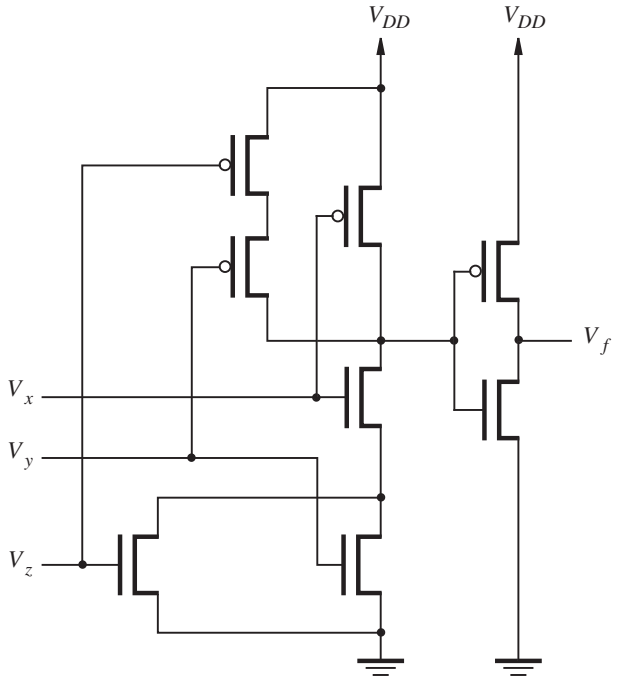


El número de transistores necesarios es 16.

3.8. El circuito completo es



3.12.



3.14. *a*) $I_D = 800\ \mu$A *b*) $I_D = 78\ \mu$A

3.17. $R_{DS} = 947\ \Omega$

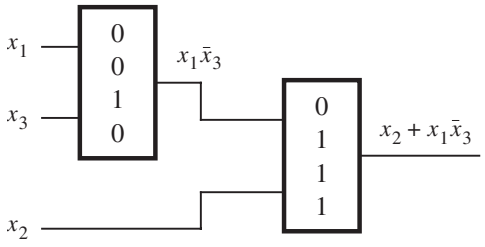
3.25. *a*) $NM_H = 0.5$V $NM_L = 0.7$ V *b*) $V_{OL} = 0.8$ V $NM_L = 0.2$ V

3.28. *a*) $P_{\text{NOT_gate}} = 163\ \mu$W *b*) $P_{\text{total}} = 8.2$ W

3.32. Los dos transistores NMOS en una compuerta NOR CMOS están conectados en paralelo. El peor caso actual para manejar la salida baja ocurre cuando sólo uno de estos transistores se pone en "ON". Por tanto, cada transistor debe tener las mismas dimensiones que el transistor NMOS en el inversor, en concreto, $W_n/L_n = 2$.

Los dos transistores NMOS están conectados en serie. Si cada uno de ellos tuviera la razón $W_p/L_p = 2$, entonces los dos transistores podrían considerarse un transistor con una razón $W_p/2L_p$. De esta manera, cada transistor PMOS debe tener el doble de ancho que en el inversor, en concreto, $W_n/L_n = 8$.

3.45. $f = x_2 + x_1\bar{x}_3$. El circuito correspondiente es



3.55. El circuito de la figura P3.11 es una compuerta XOR de dos entradas. Este circuito presenta dos inconvenientes: cuando las dos entradas son 0 el transistor PMOS debe poner f en 0, lo que da como resultado $f = V_T$ voltios. Además, cuando $x_1 = 1$ y $x_2 = 0$, el transistor NMOS debe operar la salida alta, que resulta en $f = V_{DD} - V_T$.

CAPÍTULO 4

4.1. Forma SOP: $f = \bar{x}_1x_2 + \bar{x}_2x_3$
Forma POS: $f = (\bar{x}_1 + \bar{x}_2)(x_2 + x_3)$

4.2. Forma SOP: $f = x_1\bar{x}_2 + x_1x_3 + \bar{x}_2x_3$
Forma POS: $f = (x_1 + x_3)(x_1 + \bar{x}_2)(\bar{x}_2 + x_3)$

4.5. Forma SOP: $f = \bar{x}_3\bar{x}_5 + \bar{x}_3x_4 + x_2x_4\bar{x}_5 + \bar{x}_1x_3\bar{x}_4x_5 + x_1x_2\bar{x}_4x_5$
Forma POS: $f = (\bar{x}_3 + x_4 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5)(x_2 + \bar{x}_3 + \bar{x}_4)(x_1 + x_3 + x_4 + \bar{x}_5)(\bar{x}_1 + x_2 + x_4 + \bar{x}_5)$

4.9. $f = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4$

4.11. La instrucción es falsa. Como ejemplo de contador considere $f(x_1, x_2, x_3) = \sum m(0, 5, 7)$. Por tanto, la forma SOP de costo mínimo $f = x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ es única. Pero hay dos formas POS de costo mínimo:
$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(x_1 + \bar{x}_2)$ y
$f = (x_1 + \bar{x}_3)(\bar{x}_1 + x_3)(\bar{x}_2 + x_3)$

4.12. En un circuito combinado:
$f = \bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1x_2x_3$
$g = \bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_4$
Los primeros tres términos producto son compartidos, por consiguiente el costo total es 31.

4.14. $f = (x_3 \uparrow g) \uparrow ((g \uparrow g) \uparrow x_4)$ donde $g = (x_1 \uparrow (x_2 \uparrow x_2)) \uparrow ((x_1 \uparrow x_1) \uparrow x_2)$

4.15. $\bar{f} = (((x_3 \downarrow x_3) \downarrow g) \downarrow ((g \downarrow g) \downarrow (x_4 \downarrow x_4))$ donde
$g = ((x_1 \downarrow x_1) \downarrow x_2) \downarrow (x_1 \downarrow (x_2 \downarrow x_2))$. Por tanto, $f = \bar{f} \downarrow \bar{f}$.

4.18. $f = \bar{x}_1(x_2 + x_3)(x_4 + x_5) + x_1(\bar{x}_2 + x_3)(\bar{x}_4 + x_5)$

4.21. $f = g \cdot h + \bar{g} \cdot \bar{h}$, donde $g = x_1x_2$ y $h = x_3 + x_4$

4.23. $f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_2x_3x_4$

4.32. Representando ambas funciones del mapa de Karnaugh, es fácil mostrar que $f = g$.

CAPÍTULO 5

5.1. *a*) 478 *b*) 743 *c*) 2025 *d*) 41567 *e*) 61680

5.2. *a*) 478 *b*) −280 *c*) −1

5.3. *a*) 478 *b*) −281 *c*) −2

5.4. Los números se representan como sigue:

Decimal	Signo y magnitud	Complemento 1	Complemento 2
73	000001001001	000001001001	000001001001
1906	011101110010	011101110010	011101110010
−95	100001011111	111110100000	111110100001
−1630	111001011110	100110100001	100110100010

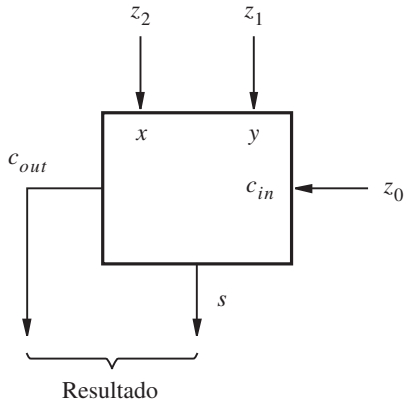
5.11. Sí, funciona. La compuerta NOT que produce c_i no es necesaria en las etapas donde $i > 0$. La desventaja es la propagación "pobre" de $\bar{c}_i = 1$ a través del transistor NMOS. El aspecto positivo es que se requieren menos transistores para producir $\bar{c}_{i+1}$.

5.12. De la expresión 5.4, cada c_i requiere i compuertas AND y una compuerta OR. Por consiguiente, para determinar todas las señales c_i necesitamos $\sum_{i=1}^{n}(i + 1) = (n^2 + 3n)/2$ compuertas. Además de esto, necesitamos $3n$ compuertas para generar todas las funciones g, p y s. Por tanto, se requiere un total de $(n^2 + 9n)/2$ compuertas.

5.13. 84 compuertas.

5.17. El código de la figura P5.2 representa un multiplicador. Multiplica los dos bits inferiores de *Input* por los dos bits superiores de *Input,* produciendo la salida *Output* de cuatro bits.

5.21. Puede utilizarse un circuito sumador completo, de tal manera que dos de los bits del número estén conectados como las entradas *x* y *y,* mientras que el tercer bit está conectado como el acarreo de entrada. Por ende, los bits del acarreo de salida y de suma indicarán cuántos bits de entrada son iguales a 1.

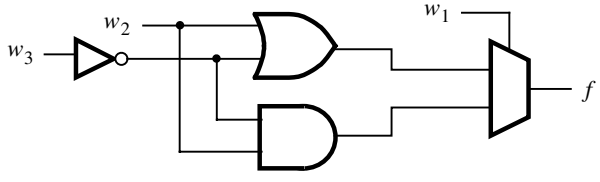


CAPÍTULO 6

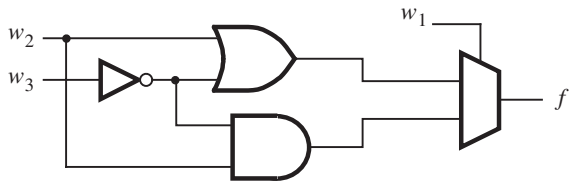
6.3.

w_1	w_2	w_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

w_1	f
0	$w_2 + \bar{w}_3$
1	$w_2\bar{w}_3$



6.5. El circuito derivado es



6.10. $f(w_1, w_2, \ldots, w_n) = [w_1 + f(0, w_2, \ldots, w_n)] \cdot [\overline{w}_1 + f(1, w_2, \ldots, w_n)]$

6.12. La expansión de f en términos de w_2 da

$$\begin{aligned} f &= \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3) \\ &= w_2 \oplus (\overline{w}_1 + \overline{w}_3) \\ &= w_2 \oplus \overline{w_1 w_3} \end{aligned}$$

El costo de este circuito es 2 compuertas + 4 entradas = 6.

6.14. Cualquier número de funciones de cinco variables puede implementarse utilizando dos LUT 4. Por ejemplo, si colocamos en cascada las LUT 4 al conectar la salida de una de ellas a una entrada de la otra, entonces podemos producir cualquier función de la forma

$$\begin{aligned} f &= f_1(w_1, w_2, w_3, w_4) + w_5 \\ f &= f_1(w_1, w_2, w_3, w_4) \cdot w_5 \end{aligned}$$

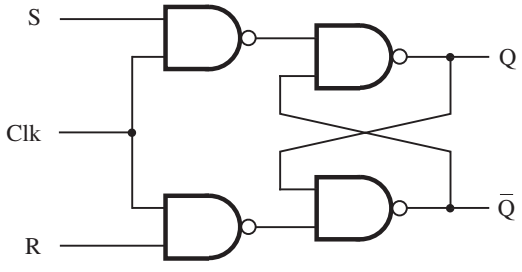
6.18. El código de la figura P6.2 es un decodificador dos a cuatro con una entrada enable. No es un buen estilo para definir este decodificador. El código no es fácil de leer. Es mejor utilizar el estilo de las figuras 6.30 o 6.46.

6.29.

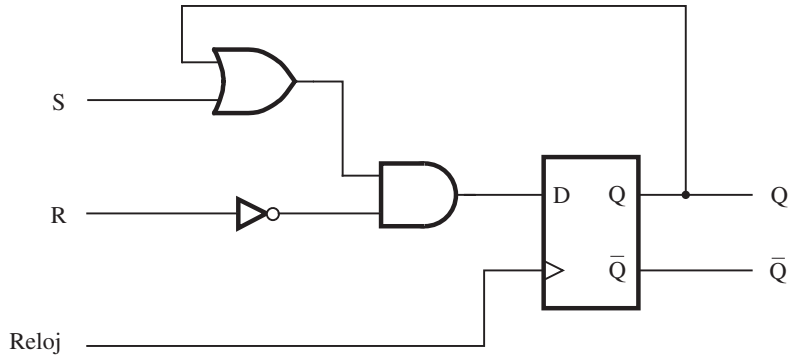
$$\begin{aligned} a &= w_3 + w_2 w_0 + w_1 + \overline{w}_2 \overline{w}_0 \\ b &= \overline{w}_1 \overline{w}_0 + w_1 w_0 + \overline{w}_2 \\ c &= w_2 + \overline{w}_1 + w_0 \end{aligned}$$

CAPÍTULO 7

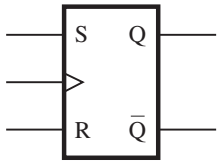
7.4.



7.6.

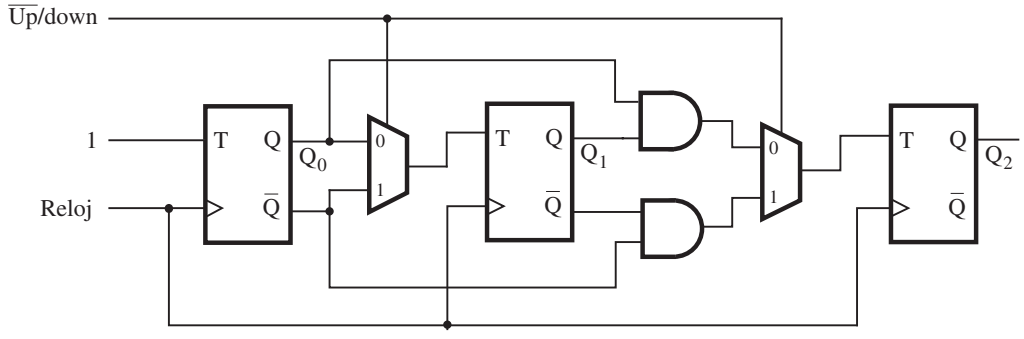


S	R	Q($t+1$)
0	0	Q(t)
0	1	0
1	0	1
1	1	0



7.9. El circuito actúa como un flip-flop JK activado por flanco negativo, en el que $J = A$, $K = B$, Reloj $= C$, Q $= D$ y $\overline{\text{Q}} = E$.

7.16.



7.18. La secuencia de conteo es 000, 001, 010, 111.

7.24. El retraso más largo en el circuito es el de la salida de FF_0 a la entrada de FF_3. Este retraso suma un total de 5 ns. Por tanto, el periodo mínimo para el cual operará el circuito de manera confiable es

$$T_{\text{mín}} = 5 + 1\,3 + 1 = 9\text{ns}$$

La frecuencia máxima es

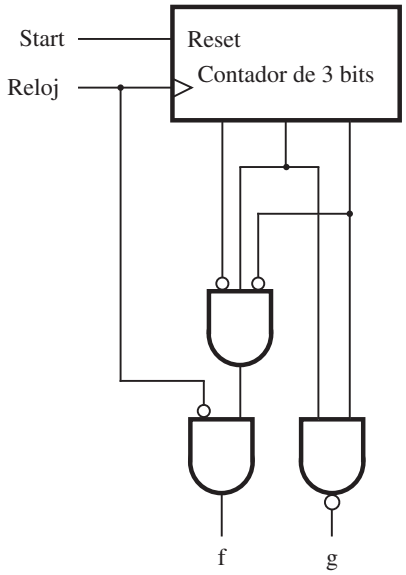
$$F_{\text{máx}} = 1/T_{\text{mín}} = 111\text{ MHz}$$

7.28.
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY prob7_28 IS
    PORT ( Clock, Reset  : IN      STD_LOGIC ;
           Data          : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Q             : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END prob7_28;

ARCHITECTURE Behavior OF prob7_28 IS
BEGIN
    PROCESS ( Clock, Reset )
    BEGIN
        IF Reset = '1' THEN
            Q <= "0000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= Q + Data ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

7.35.



CAPÍTULO 8

8.1. Las expresiones para las entradas de los flip-flops son

$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_1\overline{y}_2$$
$$D_1 = Y_1 = w \oplus y_1 \oplus y_2$$

La ecuación de salida es $z = y_1 y_2$.

8.2. Las expresiones para las entradas de los flip-flops son

$$J_2 = \overline{y}_1$$
$$K_2 = w$$
$$J_1 = \overline{w}y_2 + w\overline{y}_2$$
$$K_1 = J_1$$

La ecuación de salida es $z = y_1 y_2$.

8.5. La tabla de estado mínimo es

Estado presente	Estado siguiente		Salida z
	$w = 0$	$w = 1$	
A	A	B	0
B	E	C	0
C	D	C	0
D	A	F	1
E	A	F	0
F	E	C	1

8.5. La tabla de estado mínimo es

Estado presente	Estado siguiente		Salida z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	D	C	0	0
C	D	C	1	0
D	A	B	0	1

8.12. La tabla de estado mínimo es

Estado presente	Estado siguiente		Salida p
	$w = 0$	$w = 1$	
A	B	C	0
B	D	E	0
C	E	D	0
D	A	F	0
E	F	A	0
F	B	C	1

8.15. Las expresiones del estado siguiente son

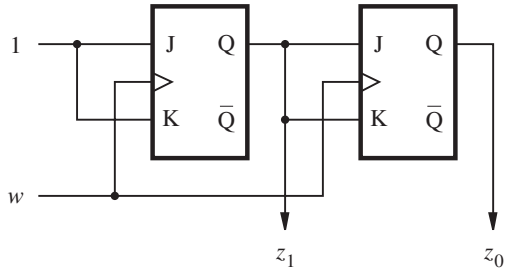
$$D_4 = Y_4 = \overline{w}y_3 + wy_1$$
$$D_3 = Y_3 = \overline{w}(y_1 + y_4)$$
$$D_2 = Y_2 = \overline{w}y_2 + wy_4$$
$$D_1 = Y_1 = w(y_2 + y_1)$$

La salida está dada por $z = y_4$.

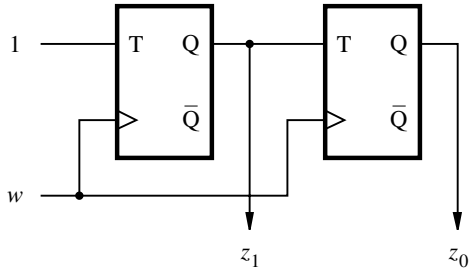
8.17. La tabla de estado mínimo es

Estado presente	Estado siguiente		Salida z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	C	0	0
C	F	C	0	1
F	C	A	0	1

8.21. El circuito deseado es



8.22. El circuito deseado es



8.29.

Estado presente	Estado siguiente		Salida
	$w = 0$	$w = 1$	z
A	A	C	0
B	A	D	1
C	A	D	0
D	A	B	0

El circuito produce $z = 1$ siempre que la secuencia de entrada en w se compone de 0 seguido por un número par de 1.

CAPÍTULO 9

9.1. La tabla de flujo es

Estado presente	Estado siguiente				
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	D	C	D	C	11
B	D	D	(B)	(B)	10
C	D	(C)	D	(C)	01
D	(D)	C	B	C	00

El comportamiento es el mismo que se describió en la tabla de flujo de la figura 9.21*a*, si se hacen los intercambios de estado A $\leftrightarrow$ D y B $\leftrightarrow$ C.

9.8. Usando el diagrama de fusión de la figura 9.40*a*, la FSM de la figura 9.39 se vuelve

Estado presente	Estado siguiente				Salida z
	$w_2w_1 = 00$	01	10	11	
A	(A)	G	E	–	0
B	(B)	C	(B)	D	0
C	B	(C)	E	(C)	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
G	B	(G)	–	D	1

9.10. La implementación libre de riesgos de costo mínimo es

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1x_3x_4$$

9.12. La implementación POS libre de riesgos de costo mínimo es

$$f = (x_1 + x_2 + x_4)(x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_3 + \bar{x}_4)(x_2 + \bar{x}_3 + x_4)$$

9.14. Si $A = B = D = E = 1$ y C cambia de 0 a 1, entonces f cambia $0 \rightarrow 1 \rightarrow 0$ y g cambia $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$. Por consiguiente, hay un riesgo estático en f y un riesgo dinámico en g.

9.17. La tabla de excitación es

Estado presente y	Estado siguiente				Salida			
	$wc = 00$	01	10	11	00	01	10	11
	Y				z			
0	(0)	(0)	1	(0)	0	0	0	0
1	0	(1)	(1)	(1)	0	1	0	1

La expresión del estado siguiente es $Y = w\bar{c} + cy + wy$. Nótese que el término wy se incluye para evitar un riesgo estático.

La expresión de salida es $z = cy$.

CAPÍTULO 11

11.1. Un juego de pruebas mínimo debe incluir las pruebas $w_1w_2w_3 = 011$, 101 y 111, así como uno de 000, 010 o 100.

11.3. Las dos funciones difieren sólo en el vértice $x_1x_2x_3x_4 = 0111$. Por tanto, los circuitos pueden distinguirse al aplicar esta combinación de entrada.

11.5. Las pruebas son $w_1w_2w_3w_4 = 1111$, 1110, 0111 y 1111.

11.9. No puede detectar si el cable de entrada w_1 está fijo en 1. La razón es que este circuito es sumamente redundante. Produce la función $f = w_3(\overline{w}_1 + \overline{w}_2)$, la cual puede implementarse con un circuito más simple.

11.11. Conjunto de pruebas = {0000, 0111, 1111, 1000}. Funcionaría con XOR implementadas como se muestra en la figura 4.28*c*.

Para *n* bits, pueden utilizarse los mismos patrones; por tanto

$$\text{Conjunto de pruebas} = \{00\ldots00, 011\ldots1, 11\ldots1, 100\ldots0\}.$$

11.12. En el circuito decodificador de la figura 6.16*c* las cuatro compuertas AND están habilitadas sólo si la señal *En* está activa. El conjunto de pruebas requerido ha de incluir las cuatro combinaciones de w_1 y w_2 cuando $En = 1$. También es necesario probar si el cable *En* está fijo en 1, lo cual puede lograrse con la prueba $w_1w_2En = 000$. Por tanto, un conjunto de pruebas completo se compone de $w_1w_2En = 000$, 001, 011, 101 y 111.

ÍNDICE ANALÍTICO

A

absorción, propiedad, 30
acarreo, 250
acarreo de entrada, 250
acarreo de salida, 252
acoplamiento cruzado, compuertas con, 383
activa baja, señal, 133
acumulador, 424, 813
alambrado, complejidad, 190
alfanuméricos, caracteres, 303
álgebra booleana, 29-37
algoritmo, 673
algoritmo D, 733
Altera APEX 20K, 909
Altera Cyclone, 911
Altera FLEX 10K, 904
Altera MAX 7000 CPLD, 902
Altera Stratix FPGA, 909
Altera Stratix II FPGA, 911
Altera UP-1 board, K95
análisis de firma, 747
análisis de temporización, 769
análisis estático de tiempos (*véase* analizador de tiempos)
análisis, 27, 196, 551, 582
Analizador de tiempos, 873
analizador lógico, 27, 751
AND, compuerta (*véase* compuertas)
AND, plano, 94
and-or-inversor, celdas, 146
ánodo, terminal, 461
apagado, transistor, 114
árbol H, 714
aritmética:
  (*Véase también* suma; división; Multiplicación; resta)
  de punto flotante (*véase* punto flotante)
  desbordamiento, 269, 760
  operadores (VHDL), 362
arquitectura (VHDL), 61, 782
  cuerpo, 782
  parte declarativa, 782
arreglo (VHDL), 780
arreglo de compuerta, 112
arreglo de compuertas de campos programables (FPGA), 5, 105
arreglo lógico programable (*véase* PLA)
arreglo multiplicador (*véase* multiplicación)
ASCII, código, 302
ASIC, 6, 11
asignación aritmética (VHDL), 285
asignación de señal seleccionada (VHDL), 299, 340, 791
asignación de señal simple, 62, 789
asignación de variables, operador, 802
asignaciones de pines, 889
asociación por nombre (VHDL), 282, 786
asociación posicional (VHDL), 282, 786
atajos de teclado, 841
atascamiento, falla, 726
atributo (VHDL), 420, 509, 506
axiomas de álgebra booleana, 29
ayuda sensible al contexto, 830, 857

B

base, 246, 870
BCD (*véase* decimal codificado en binario)
bibliotecas, 55, 224
  ieee, 283
  work, 284, 785
BILBO (observador de bloques lógicos integrado), 745
BIST (pruebas interconstruidas), 741
bit, 247
BIT, tipo, 60, 775
bloque ASM, 558
bloque de arreglo lógico (LAB), 903, 904, 909
bloque lógico, 105
bloque lógico configurable (CLB), 907
bloques de arreglos embebidos (EAB), 904
bloques SRAM en PLD, 673
borrado asíncrono (en VHDL), 422, 808
borrado asíncrono (reset), 393, 412
borrado síncrono (reset), 393, 410, 423
Boundary Scan, 748
buffer, 131, 463
  inversor, 131
  triestado, 93, 442
  VHDL (modo del puerto), 434
bus, 435, 441, 870
byte, 247

C

CAD (*véase* diseño asistido por computadora)
cadena de acarreo, 408, 762, 769, 877
canal (en MOSFET), 115
capacitancia, 121, 147
capacitancia parásita, 121
capacitor de desacoplamiento, 749
captura esquemática, 55, 278
característica de transferencia de voltaje (VTC), 119
carga de entrada, 187, 128
carga de salida, 130
carpeta, 828
carta ASM (*véase* máquina algorítmica de estados)
CASE, instrucción, 356, 457, 797
cátodo, terminal, 461
celdas de almacenamiento, 107
celdas estándar, 111, 146
chips a la medida, 6, 111
chips estándar, 4
ciclo de trabajo, 883
circuito árbitro, 543, 597
circuito compresor, 743
circuito de control, 664
circuito lógico, 2, 26
circuitos combinacionales, 315-372
circuitos secuenciales asíncronos (*véase* circuitos secuenciales)
circuitos secuenciales síncronos (*véase* circuitos secuenciales) síntesis, 27, 39, 56, 321, 488, 590, 840
circuitos secuenciales, 480
  análisis, 551, 582
  asignación de estados, 483, 491, 618
  asignación de estados en VHDL, 509
  asíncronos, 578-657
  definición de, 380, 480
  diagrama de estado, 482
  diagrama de fusión, 607
  diagrama de transición, 621
  máquina de estado finito, 480
  minimización de estados, 524-531, 603-617
  modelo formal, 559
  pruebas, 737-748

síncronos, 380, 480-560
tabla de estados, 483
tabla de flujo, 581
clear, entrada (*véase* reset, entrada), 393, 406
clock (entrada), función enable, 664
Clock' EVENT, 420
CMOS, tecnología, 78
cobertura, 174
falla de, 729
mínima, 209
tabla de, 209
codificación 1 activo, 330, 414, 494, 633
codificador binario (*véase* Codificador)
codificador, 336
en código de VHDL, 345, 353
codificador:
binario, 335, 446
de prioridad, 336
código:
BCD (*véase* decimal codificado en binario)
binario, 247
convertidor, 337, 366
decimal, 246
detección de errores, 303
Gray, 366
códigos de caracteres, 301
código de VHDL, 434, 454, 814
código de VHDL estructural, 281
código de VHDL jerárquico, 282-284, 343, 349, 786
código por comportamiento VHDL de alto nivel, 466
cofactor, 326
colector abierto, 915
colocación, 748, 767
combinación, propiedad, 31
comentario (VHDL), 774
comparador, 238
complemento:
a 1, 258
a 2, 259
a la base disminuido, 268
a la base, 265
de una variable lógica, 23
complemento a 1, representación, 258
complemento a 10, 265
complemento a 2, representación, 259
complemento a 9, 265
complemento a base, 265
complemento a la base disminuido, 268
componentes (VHDL), 281, 445, 666, 786
comportamiento del código de VHDL, 285, 339-342, 466, 802
comportamiento funcional, 27
compuertas, 25
AND, 26
NAND, 45, 80, 83
NOR, 45, 80, 85
NOT, 26, 78, 82
OR, 26
XNOR, 254, 135
XOR, 252, 135
compuerta (en el transistor MOSFET), 76
compuerta compleja (CMOS), 86
compuerta de transmisión, 134, 136
compuerta flotante, 138
compuerta, optimización, 758
compuertas lógicas, 25
capacidad de manejo, 131
característica de transferencia, 120
carga de entrada, 187, 128, 863, 863
carga de salida, 130
disipación de potencia, 124
margen de ruido, 119, 914
operación dinámica, 121
retardo de propagación, 122
tiempo de caída, 123
tiempo de elevación, 122
computadora, 9
con signo, 256
concatenación (VHDL), 287, 362, 790
condición de carrera, 592
configuración de chip, 58
conjunto de cortes, 590
conjunto de pruebas, 727-737
conjunto-DC, 220
conjunto-ON, 220
conmutativa, propiedad 30
consenso, propiedad, 31
consistencia, revisión de, 731
constante (en VHDL), 779
contador:
ascendente, 402, 676
ascendente, 403
ascendente/descendente, 403
asíncrono, 403
BCD, 412
capacidades enable y clear, 406
carga en paralelo de, 408
diseño de circuito asíncrono, 595
diseño de, 415
en anillo, 414
en cascada, 403
inicialización, 410, 412
Johnson, 415, 468
módulo n, 468
síncrono, 405, 406, 533
contador ascendente, 402, 432, 454, 676
contador ascendente/descendente, 403
contador asíncrono, 403
contador descendente, 403, 434
contador en anillo, 414
contador en cascada, 403
contador síncrono, 405, 406
conteo de bits, circuito de, 675, 802
conversión de números, 247-250
conversión de tipos (VHDL), 780
convertidor de código BCD a siete segmentos, 338, 358
convertidor de paralelo a serial, 569
convertidos de serial a paralelo, 400
corriente de fuga, 119
costo, 41, 174
cruzado, interruptor 319

D

datos, 334
decimal codificado en binario (BCD), 297
contador, 463
dígitos, 297
suma, 297
decimales, números, 246
decodificador, 329, 451
árbol, 331
decodificador BCD a 7 segmentos, 338, 463
decodificador binario (*véase* Decodificador)
DeMorgan, teorema de, 31
Demultiplexer, 333
desbordamiento (*véase* aritmética, desbordamiento)
descomposición (*véase* descomposición funcional)
descomposición disyunción, 193
descomposición funcional, 190
descomposición no disyunción, 193
desviación (*véase* desviación de reloj)
desviación del reloj, 437, 713
detector de secuencias, 481, 561
diagrama de fusión, 607
diagrama de tiempo, 27,486
diagrama de Venn, 33-36
DIP, paquete, 91
dirección, 334
directorio, 828
diseño asistido por computadora (CAD), 54-58
análisis de tiempo, 769
configuración de chip, 895
herramientas, 54
ingreso del diseño, 54
mapeo de tecnología, 760
simulación de tiempo, 57, 871
simulación funcional, 57, 846
diseño de exploración sensible al nivel, 741
diseño físico, 57
diseño jerárquico, 55, 274
diseño libre de riesgos, 638
diseño para aplicación de pruebas, 737
disipación de potencia, 124
dinámica, 124, 126
en circuitos CMOS, 125
en circuitos NMOS, 124
estática, 124, 127, 144
PRBSG, 743

disparado por flanco, 388, 392
dispositivo lógico programable (*véase* PLD)
dispositivo lógico programable complejo (CPLD), 101
distributiva, propiedad, 30
divide y vencerás, 12
división, 686
divisor de reloj, 461
doble precisión (*véase* punto flotante)
dominancia de columna, 211
dominancia de renglón, 207
drenado abierto, 915
drenado (en transistor MOSFET), 76
dualidad, 30

E

EDA, herramientas, 830
edición (Waveform), herramienta de, 845
EDIF, 833
editor de texto (Quartus II), 848
efecto cuerpo, 127
elemento de exclusión mutua, 602
elemento lógico, 904
elemento sensible al nivel, 388
elementos de almacenamiento volátiles, 109
eliminación de rebotes, 718
enable, entrada, 406, 518, 664
energía (capacitor), 125
enrutamiento, 748, 768
  canal, 105, 111
enteros:
  con signo, 246
  en VHDL, 289, 433, 778
  sin signo, 246
ENTITY, 60, 781
ENTITY, declaración, 60, 782
  con el parámetro GENERIC, 793
ENTITY, diseño (véase ENTITY)
entradas asíncronas, 717
enum_encoding, 509
enum_encoding, codificación, 509
ENUMERATION, tipo (VHDL), 778
equivalencia funcional, 28
equivalencia:
  de estados, 524
  de redes lógicas, 28
errores en el código de VHDL, 821
esclavo (véase Flip-flop, maestro-esclavo)
Espresso, 223
esquema, 25
estado, 480
  asignación, 483, 491, 618
  asignación en VHDL, 509
  compatibilidad, 607
  definición de, 380, 480
  diagrama, 482
  equivalencia, 524
  minimización, 524-531, 603
  tabla, 483
  variables, 483, 579
estado actual, 483, 579
  variables, 484, 579
estado estable, 578
estado inestable, 587
estado inicial, 481
estado siguiente, 483, 579
  variables, 484, 579
estados adyacentes, diagrama, 621
estados asignados, tabla, 484
estados compatibles, 607
estándares:
  1076 (VHDL), 58
  1149.1 (pruebas), 748
  1164 (VHDL), 58
  punto flotante del IEEE, 295
  Verilog, 55
estructura de árbol, 733
EVENT, 420, 806
EVENT, atributo, 420
Excess-1023, formato, 296
Excess-127, formato, 296
exploración de ruta, 738
expresiones canónicas:
  producto canónico de sumas, 43
  suma canónica de productos, 41
expresiones lógicas, 21
extensores paralelos, 903

F

fabricados a la medida (custom chips), 111
factorización, 186
fallas:
  de atascamiento, 726
  detección, 727, 731
  modelo, 726
  propagación, 731
fiabilidad, 751
firma, 743
flanco (en señales), 389
flanco activo del reloj, 389,480
flanco negativo, 389
flip-flop, 389, 398
flip-flop D (véase flip-flop)
flip-flops:
  código de VHDL para, 420, 806
  configurables (en PLD), 397
  D, 389, 420
  disparados por flanco, 392, 398
  disparados por flanco negativo, 392
  disparados por flanco positivo, 392, 395
  entradas clear y preset, 393
  JK, 398, 467, 536
  maestro-esclavo, 389, 398, 582
  SR, 369
  T, 394
Floorplan Editor, 863
flujo de corriente:
  de compuerta, 116
  de cortocircuito, 125, 133
  de fuga, 119
  dinámico, 123
  estático, 119, 127, 144
  hundirse, 461
fmax, 770, 885
FOR GENERATE, instrucción, 348, 432, 454, 683, 692, 703, 793
FOR LOOP, instrucción, 432, 442, 797, 801
FPLA (*véase* PLA)
FSM completamente especificada, 531
FSM especificada de manera incompleta, 531
FSM (*véase* máquina de estado finito)
fuente (en transistor MOSFET), 76
función original, 237
funciones especificadas de manera incompleta, 180
funciones lógicas, 21
  AND, 22
  minimización, 172-179, 207-222
  NAND, 45. 80
  NOR, 45
  NOT, 23
  OR, 22
  síntesis, 37-44
  XNOR, 254
  XOR, 252
fusibles, 136
fusión, 605
  procedimiento, 607

G

generación de pruebas, 727-737, 842
generador de secuencias binarias pseudoaleatorias (PRBSG), 742
generados de paridad serial, 591
GENERATE, instrucción, 348, 454, 793
GENERIC (*véase* ENTITY, declaración)
GENERIC MAP, 426, 428, 815
Gray, código, 366

H

Hamming, distancia, 618
hardware digital, 2
herramientas (CAD), 758
heurístico, método, 176
hexadecimales, números, 248
hipercubo, 207
Huntington, postulados de, 31

I

IEEE, 55
IEEE, estándares (*véase* estándares)
IF GENERATE, instrucción, 349
IF, instrucción, 350, 796

impedancia característica, 750
implicante primo, 174
implicante primo esencial, 175, 218
implicante, 173
inanición, 551
ingreso del diseño, 54
inicialización de memoria, archivo de, 906
instanciación (de componentes VHDL), 281, 445, 785
instrucción de asignación concurrente (VHDL), 350, 788
instrucción de asignación secuencial (VHDL), 344, 792
instrucción de asignación secuencial (VHDL), 350, 794
instrucción de proceso (VHDL), 350, 794, 798
instrumentación, 751
INTEGER, tipo (VHDL), 289, 778
integración de escala pequeña (SSI), 93
integración de gran escala (LSI), 93
integración de mediana escala (MSI), 93
integración de muy gran escala (VLSI), 93
interferencia, 749
interruptor programable, 4
interruptor, 20
intersección, 33
inversión, 23
inversor, 78, 82

J
JK, flip-flop, 398, 467, 536
Johnson, contador, 415, 468
JTAG, puerto, 104

K
Karnaugh, mapa de, 164-172
k-cubo, 207

L
latch:
  código de VHDL, 419
  D, 398
  D asíncrono, 386, 398, 419, 582
  SR asíncrono, 383, 385, 398
  SR básico, 381, 398, 578
  SR con set dominante, 472
latch asíncrono, 385, 398
latch básico, 381, 398
latch D asíncrono, 398, 419, 582, 805
latch SR (véase latch)
latch SR asíncrono, 383, 385, 398
LED (diodo emisor de luz), 132, 460
lenguaje de descripción de hardware (HDL), 55
Library of Parameterized Modules (LPM), 279, 786
LPM__WIDTH, 426
lpm_add_sub, 279, 424, 876
lpm_counter, 427
LPM_DIRECTION, 426
lpm_ff, 424
lpm_ram_dq, 701, 721, 906
lpm_rom, 906
lpm_shiftreg, 426
línea de transmisión, efectos, 750
lineamientos, 835
lista de redes, generación de, 758
lista de sensibilidad (VHDL), 350, 798
literal, 173
lógica de arreglo programable (*véase* PAL)
lógica de transitor-transistor (TTL), 914
lógica mixta, 90
lógica negativa, 74, 88
lógica positiva, 74, 88
Loop, instrucción (*véase* FOR LOOP)
LUT, 107

M
macrocelda, 98, 901
macrofunción, 279
maestro (*véase* flip-flop maestro-esclavo)
maestro-esclavo (*véase* flip-flop)
magnitud, 256
mapeo de tecnología, 223, 760
máquina algorítmica de estados (ASM):
  bloque ASM, 558
  cartas ASM, 555
  casilla de decisiones, 556
  casilla de estado, 556
  casilla de salida condicional, 556
  temporización implicada, 675
máquina de estado (*véase* máquina de estado finito)
máquina de estado finito (FSM), 480
  chip del temporizador programable (555), 723
  especificada de manera incompleta, 531
  resumen del procedimiento de diseño, 488, 555
máquina expendedora, controlador, 469, 526, 642
mar de compuertas, tecnología, 112
más significativo, bit, 247
maxtérmino, 42
Mealy, FSM tipo, 480, 496
  código de VHDL, 511, 818
Mealy, salida tipo, 556
media, operación de la, 696
Mega Wizard Plug-in Manager, 876
megafunción, 279
memoria
  implícita (VHDL), 355, 419, 799
memoria de sólo lectura (ROM), 334
memoria de sólo lectura programable y borrable (EPROM), 140
memoria de sólo lectura programable y borrable eléctricamente (EEPROM), 1
memoria estática de acceso aleatorio (SRAM), 142, 668, 700
memoria implícita (VHDL), 355, 419, 455, 799
menos significativo, bit, 247
metaestabilidad, 717
métrica de compuertas equivalentes, 105
minimización:
  de estados, 524-531, 603-617
  de funciones lógicas, 172-179, 207-222
mintérmino, 40
modo (de un bloque lógico), 760
modo de pulso, 578
modo fundamental, 578
módulo lógico adaptativo (ALM), 911
Moore, FSM tipo, 480
Moore, ley de, 2
Moore, salida tipo, 556
  código de VHDL, 502, 816
MOSFET, transistor, 75
  con resistencia de encendido, 117
movimiento de banda de goma, 839
multinivel, circuitos, 185-203, 864
multiplexor (código de VHDL), 341-344, 446
multiplexor (definición), 333
multiplexor, 51, 316-328, 133, 136
multiplicación, 289, 291, 677
  arreglo multiplicador, implementación, 291
  implementación secuencial, 677
  operando con signo, 292
  producto parcial, 290

N
NAND, circuitos, 45-47, 195,80
NAND, compuerta (*véase* compuertas)
n-cubo, 207
nibble, 247
nivel de transferencia de registros (RTL), código, 466
nivel de voltaje
  alto, 74
  bajo, 74
  polarización del sustrato, 127
  V_{IH} y V_{IL}, 120
  V_{OH} y V_{OL}, 120
NMOS, tecnología, 78
NMOS, transistor, 75
Node Finder, 844
nodo (Quartus II), 838
no-importa, condición, 180
  en código de VHDL, 358
nombres (VHDL), 774

NOR exclusivo (XNOR), compuerta (*véase* compuertas)
NOR, circuitos, 45-48, 195, 80
NOR, compuerta (*véase* compuertas)
NOR, plano, 138
NOT, compuerta (*véase* compuertas)
números (en VHDL), 774
números binarios, 247
  en código de VHDL, 687
números con punto fijo, 293
números negativos, 256
números sin signo, 246

O
octales, números, 248
Odd function, 252
operación coincidencia, 254
operaciones (*véase* funciones lógicas)
operación-Sharp (operación-#), 218
operadores (VHDL), 359, 781
operadores de corrimiento (VHDL), 362
operadores lógicos (VHDL), 259
operadores relacionales (VHDL), 361
optimización (*véase* minimización)
OR exclusivo (XOR), compuerta (*véase* compuertas)
OR, compuerta (*véase* compuertas)
OR, plano, 94
orden de instrucciones (VHDL), 350, 431, 799, 812
ordenación, operación, 703
ordenamiento de instrucciones (VHDL), 350, 431, 799, 812
oscilador en anillo, 475
osciloscopio, 27, 751
OTHERS (VHDL), 340, 428, 790

P
PAL, 97
palabras reservadas (VHDL), 774
pantalla 7 segmentos, 338
paquete (VHDL), 225, 283, 444, 454, 784
paquete BGA, 106
paquetes (físicos):
  arreglo de pines en retícula (PGA), 106
  ball grid array (BGA), 106
  en línea doble (DIP), 91
  plastic-leaded chip carrier (PLCC), 100
  quad flat pack, 102
  small-outline integrated circuit (SOIC), 93
parámetro de transconductancia del proceso, 116
paridad, 303, 591
patas (pines), 91
pines, 91
pinstub, 838
PLA, 94, 138
plantilla (arreglo de compuerta), 112
plantillas (VHDL), 849
PLD, 5, 94
PMOS, transistor, 76
p-n, unión, 114
polisilicón, 114
PORT MAP, 282, 786
portabilidad, 55
precedencia de las operaciones, 37, 363
precisión sencilla (*véase* punto flotante)
preset, entrada, 393
primitives, biblioteca, 833
prioridad, 336
problema técnico, 587, 634
procesador, 449
proceso de diseño, 6
producto de sumas, forma (POS), 43
producto estrella, operación (operación-*), 216
producto lógico (AND), 36
producto parcial, 290
productos de sumas (SOP), forma, 40
programación en el sistema (ISP), 100, 902
programación no volátil, 104
propiedad asociativa, 30
propiedades del álgebra booleana, 30
proyecto (Quartus II), 828
pruebas, 506, 726, 727-752
pruebas aleatorias, 734
pruebas pseudoaleatorias, 742
pseudo-NMOS, tecnología, 119, 149
puerto (VHDL), 61, 782
punto flotante, 295
  de doble precisión, 295
  estándar IEEE, 295
  exponente, 295
  formato de, 295
  mantisa, 295
  normalizada, 295
  precisión sencilla, 295
  representación, 295

Q
QFP, paquete, 102
Quartus, archivo de proyecto, 860
Quine-McCluskey, método, 207

R
RAM (*véase* memoria estática de acceso aleatorio)
ramificación heurística, 213, 221
razón precio/rendimiento, 270
red, 25
red de bajada, 81
red de subida, 82
red lógica, 26
reflexiones, 749
región de saturación, 116
región de tríodo, 116
región lineal (*véase* región de tríodo)
registro (código de VHDL), 426
registro de corrimiento de retroalimentación lineal (LFSR), 742
registro, 399
  código de VHDL, 808
registro de corrimiento, 399, 666
  código de VHDL, 430, 811
registro de corrimiento en cilindro, 369, 685
reloj, 385, 883
representación cúbica, 203-207
representación de signo y magnitud, 258
representación numérica:
  complemento a 1, 258
  complemento a 10, 265
  complemento a 2, 259
  decimal codificado en binario, 297
  en VHDL, 284
  enteros con signo, 256
  enteros sin signo, 246
  hexadecimales, 248
  notación posicional, 246
  octales, 248
  punto fijo, 293
  punto flotante, 295
  signo y magnitud, 258
reset, entrada, 380, 482
resistencia (canal del transistor), 117
resistencia de encendido, 117
resistor de subida, 79
resolución, función de (VHDL), 777
resta, 262
retardo (*véase* retardo de propagación)
retardo de compuerta (*véase* retardo de propagación)
retardo de propagación, 57, 122, 388
retroalimentación, 381
riesgo dinámico, 635
riesgo estático, 417, 634
riesgos, 634-641
  dinámicos, 635
  estáticos, 417, 634
ROM (*véase* Memoria de sólo lectura)
ROM programable (PROM), 335, 109
rotación, operadores de (VHDL), 362
rotar símbolo, 835
ruido, 119
  margen de, 119
  suministro de corriente, 749
ruta crítica, 271, 770, 873
ruta de datos, 664

S
salida de alta impedancia, 132
salida múltiple, circuitos de, 182
sección (*slice*), 912
Selection, herramienta (Waveform), 845
semáforo, controlador de, 722
semiconductor complementario de metal-óxido (*véase* CMOS, tecnología)
semiconductor de metal-óxido (*véase* MOSFET)
semiconductor, 114
Sensibilización de ruta, 729
señal reset, estado inicial, 481
señales globales, 504, 714
señalización de reconocimiento, 597
serie 7400, chips, 91
Shannon, teorema de expansión, 324
SIA, guía, 2
SIGNAL, 775
SIGNED, tipo, 288
signo, bits, 256
signo, extensión, 293
simplificación (*véase* minimización)
simulación:
  de tiempo, 57
  funcional, 57
simulación de tiempo, 57
simulación funcional, 57
simulador, 57
sincronización del reloj, 402, 713
SIS (síntesis secuencial interactiva), 223
  lógica, 37-44
  multinivel, 185-196, 864
sistema digital, 664
socket, 101
solapamiento (*aliasing*), problema en pruebas, 745
Speed Grade, 861, 900
STD_LOGIC, tipo, 224
subtipo (VHDL), 776
sucesor *k*, 524
suma, 250-255, 261
  BCD, 297
  acarreo, 250
  desbordamiento, 269
  función generada, 271
  función propagada, 271
  suma, 250
  VHDL, 285-289
suma lógica (OR), 36
sumador;
  BCD, 297
  acarreo de adelanto, 271
  acarreo en cascada, 255, 867
  acarreo guardado, 309
  en código de VHDL, 281-289
  medio sumador, 250
  retraso de propagación, 270, 276
  serial, 514
  sumador completo, 252
sumador completo, 252, 281, 784
sumador con acarreo de adelanto, 271
sumador de acarreo en cascada, 255
sumador de acarreo guardado, 309
sumador medio, 250
sumador serial, 514
sumador/restador, 264, 459
sustrato, 76

T
T, flip-flop, 394
tabla característica, 382
tabla de consulta, 107
tabla de excitación, 581
tabla de flujo, 581
  primitiva, 605
  reducción de estados, 603-617
tabla de verdad, 24
tablas de flujo primitivas, 605
tamaño del circuito, 105
tarjeta de circuito impreso (PCB), 13, 748
tarjeta madre, 9
temporizador, 723
teorema de expansión (Shannon), 325
teoremas de álgebra booleana, 30
terceros, herramientas de, 830
terminaciones, 749
tiempo de caída, 123
tiempo de espera, 389, 770, 885
tiempo de preparación, 388, 714, 770, 885
tiempo de reloj-tiempo de salida(TCO), 417, 716, 770, 886
tiempo de retardo de salida (TOD), 715
tiempo de retardo del registro (TRD), 714
tiempo de subida, 122
tipo (VHDL), 775
tolerancia, 770
transferencia de voltaje, característica de, 119
transferencia en paralelo, 400
transición, diagrama de, 621
transición, tabla (*véase* tabla de excitación)
transistor de paso, 144
transistor:
  EEPROM, 138
  EPROM, 140
  MOSFET, 75
tamaño, 123
triestado:
  buffer, 93, 132
  código de VHDL, 442
triestado, salida (*véase* triestado)
túnel Fowler-Nordheim, 139

U
ultravioleta, luz, 140
unidad lógica aritmética (ALU), 358
unión, 33
universal, registro de corrimiento, 473
UNSIGNED, tipo, 288
USE, cláusula, 224, 784

V
valor predeterminado (VHDL), 455, 799
valoración, 24
valores lógicos, 20
variable binaria, 20
variable de entrada, 21
VARIABLE, 779, 800
vectores de prueba (*véase* generación de pruebas)
Verilog HDL, 55
vértice, 204
VHDL, 58-63, 224, 281-288, 339-363, 502-512, 773-824
  arquitectura, 61, 782
  arreglo, 705, 780
  asignación aritmética, 285
  asignación de señal condicional, 344, 792
  asignación de señal seleccionada, 299, 340, 791
  asociación por nombre, 282
  asociación posicional, 282
  borrado asíncrono, 422, 808
  BUFFER, 434
  CASE, instrucción, 356, 457, 797
  comentarios, 774
  componente, 281,786
  concatenación, 287, 790
  entidad, 60, 782
  FOR LOOP, 432, 797, 801
  GENERATE, 348, 454, 793
  IF, instrucción, 350, 796
  instanciación de componentes, 281, 445, 666, 785
  lista de sensibilidad, 350, 798
  memoria implícita, 355, 419, 799
  no-importa, 358, 776
  nombres, 774
  operadores, 359, 781
  orden de instrucciones, 350, 431, 799, 812
  paquete, 225, 283, 444, 454, 784
  precedencia, 363
  proceso, 350, 794, 798
  puerto, 61, 782
  representación numérica, 284, 774
  reset síncrono, 423, 808
  señal, 775
  variable, 779, 800
  vector, 775

vía, 111
voltaje umbral, 74, 114

W
WAIT UNTIL, instrucción, 420, 430, 807
Waveform Editor, 845
WHEN, cláusula (VHDL), 340, 356, 792
World Wide Web, 223

X
Xilinx Spartan-3, 913
Xilinx Virtex FPGA, 912
Xilinx Virtex II (Pro) FPGA, 912
Xilinx XC4000 FPGA, 907
XNOR (NOR exclusivo), compuerta (*véase* compuertas)
XOR (OR exclusivo), compuerta (*véase* compuertas}